AFRL-RI-RS-TR-2013-127

SCALABLE ALGORITHMS FOR PARALLEL DISCRETE EVENT SIMULATION SYSTEMS IN MULTICORE ENVIRONMENTS

THE STATE UNIVERSITY OF NEW YORK AT BINGHAMTON

MAY 2013

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2013-127 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

RYAN LULEY
Work Unit Manager

/ S /

MARK LINDERMAN, Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2013	FINAL TECHNICAL REPORT	NOV 2010 – NOV 2012

4. TITLE AND SUBTITLE

SCALABLE ALGORITHMS FOR PARALLEL DISCRETE EVENT SIMULATION SYSTEMS IN MULTICORE ENVIRONMENTS

5a. CONTRACT NUMBER
FA8750-11-2-0004

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
63788F

6. AUTHOR(S)

Dmitry Ponomarev, Nael Abu-Ghazaleh

5d. PROJECT NUMBER
T3PD

5e. TASK NUMBER
BI

5f. WORK UNIT NUMBER
NG

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The State University of New York at Binghamton
P.O. Box 6000
Vestal Park East
Binghamton, NY 13902

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2013-127

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2013-2240
Date Cleared: 9 May 2013

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project investigated techniques for improving performance and scalability of parallel discrete event simulation systems on multicore and manycore processors and clusters of multicores. Specifically, we designed and optimized a multithreaded version of ROSS simulator to efficiently execute in these environments. In addition, we also investigated efficient model partitioning schemes and also studies techniques for improving simulation resiliency to the presence of external interference. Finally, we evaluated PDES performance on the Tilera architecture and proposed optimization techniques for that environment.

15. SUBJECT TERMS
parallel discrete event simulation, multi-core algorithms, Time Warp, Tilera, optimistic simulation, shared memory communication, ROSS, model partitioning, dynamic mapping

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**RYAN LULEY**
U	U	U	SAR	116	19b. TELEPONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1.0 Project Summary

The arrival of multicore processors, and their ongoing evolution into manycores, makes parallelism the primary vehicle for improving application performance. Manycores provide an environment with ultra-low communication latencies that can substantially impact fine-grained application such as parallel discrete event simulation (PDES) whose performance and scalability are often limited by the communication cost. Traditionally, studies with PDES algorithms on Beowulf Clusters are almost singularly focused on tolerating the impact of latency because of its dominant effect on performance. Relatively less attention is paid to other aspects of the simulator because the impact of other optimizations is secondary relative to communication. As on-chip latency is significantly reduced on manycores, performance bottlenecks shift to more typical algorithmic, load-balancing and synchronization problems. In addition, the deep memory integration available on manycores enables new optimization opportunities. It is therefore important to consider the redesign of key PDES algorithms to efficiently support high performance and scalable execution on the manycore machines and their clusters.

In addition, PDES performance (and in fact, that of many parallel applications) suffers disproportionately in the presence of interference from co-located applications and/or system services. The slowdown experienced generally far exceeds expectations; for example, in one of the simulators we studied, the presence of even a single interfering process caused unbounded slowdown of an 8-way simulation. The problem occurs because in an application with dependencies, when one thread is inactive due to a context switch, the remaining threads may not be able to proceed while they wait for dependencies to resolve. In order to provide robust and high-performing PDES operation, the issues of external noise and interference have to be considered as first-class design considerations.

In this project, we performed the research that addressed some aspects of the aforementioned issues. Specifically, our investigations mainly focused along several directions, and significant new results have been achieved (and published or submitted for publication) for all of them:

- First, we implemented a thread-based version of the Rensselaer's Optimistic Simulation System (ROSS) PDES simulator. The multi-threaded implementation eliminates multiple

message copying and significantly minimizes synchronization delays. We studied the performance of the simulator on two hardware platforms: an Intel Core i7 machine and a 48-core AMD Opteron Magny-Cours system. We identified performance bottlenecks and proposed and evaluated mechanisms to overcome them. Results showed that multithreaded implementation improves performance over the MPI version by up to a factor of 3 for the Core i7 machine and 1.2 on Magny-Cours for 48-way simulation. This work was published in the International Parallel and Distributed Processing Symposium (IPDPS) in 2012.

- Second, we examined the performance of multi-threaded implementation of PDES, as described in the above paragraph, on Clusters of Multicores (CMs). We demonstrated that the inter-node communication costs impose a substantial bottleneck on PDES and showed that without optimizations addressing these long latencies, multithreaded PDES does not significantly outperform the multiprocess version despite direct communication through shared memory on the individual nodes. We then proposed three optimizations: message consolidation and routing, infrequent polling, and latency-sensitive model partitioning. We showed that with these further optimizations in place, threaded implementation of PDES significantly outperforms process-based implementation even on CMs. Our paper based on this material has been accepted to the ACM SIGSIM-PADS 2013 Conference.

- Third, we performed extensive evaluation of PDES on Tile64Pro - a 64-core chip from Tilera that was provided to us by AFRL specifically for this project. For these studies, we used the multi-threaded version of ROSS simulator and showed that the performance of this simulator (with many optimizations proposed) scales by a factor of 27X when it is executed on 56 cores of the Tilera chip for Phold benchmark with 20% remote communication. We also evaluated the impact of performance optimizations that we proposed on both conservative and optimistic versions of the simulator and also analyzed the sensitivity to various simulation parameters. Finally, we explored the issues of object placement and model partitioning on Tilera architecture. This work was published in PADS Workshop 2012.

- Fourth, we examined model partitioning algorithms for PDES. Partitioning plays an important role in PDES performance due to the high communication cost in parallel platforms and the fine-granularity of most simulation models. We explored how partitioning based on dynamic information about the simulation should be approached and explored policies that focus on communication cost, load balancing and both. We showed that on multicore clusters, dynamic partitioning achieves up to 4x better performance than static partitioning. On the AMD Magny-Cours, where the communication latency is low, dynamic partitioning results in a 2x performance improvement over static partitioning for some of our models. This work was published in PADS Workshop 2012.

- Fifth, we showed that the presence of interference from other users, even a single process in an arbitrarily large parallel environment, can lead to dramatic slowdown in the performance of the simulation. We defined a new metric, which we call proportional slowdown that represents the idealized target for graceful slowdown in the presence of interference. We identified some of the reasons why simulators fall far short of proportional slowdown. Based on these observations, we designed alternative simulation scheduling and mapping algorithms that are better able to tolerate interference. More precisely, the most resilient simulators will allow dynamic mapping of simulation event execution to processing resources (a work pool model). However, this model has significant overhead and can substantially impact locality. Thus, we proposed a locality-aware adaptive dynamic-mapping (LADM) algorithm for PDES on multi-core systems. LADM reduces the number of active threads in the presence of interference, avoiding having threads disabled due to context switching. Our paper based on this material has been accepted to the ACM SIGSIM-PADS 2013 Conference.

2.0 Introduction

Discrete Event Simulation (DES) is a type of simulation used to study systems where the changes of state are discrete; for example, it is widely used in the simulation of computer and telecommunication systems, biological networks, and war simulation. The increasing demands of simulation models challenge the capabilities of sequential simulators. PDES exploits the natural parallelism present in simulation models to substantially improve the performance and capacity of DES simulators, allowing the simulation of larger, more detailed models in shorter times.

In PDES, a simulation model is partitioned across a number of processes (called Processing Elements, or PEs). Each PE processes events in simulation time order, sending messages to remote PEs if future events are generated to them. These messages must be processed in correct simulation time to maintain causality. This global time synchronization can be supported in two ways: (1) conservative simulation requires PEs to coordinate to guarantee that no causality errors can occur (i.e., simulation time does not progress beyond a point until all events that occur prior to that point are received and processed); and (2) optimistic simulation: no explicit synchronization is enforced between PEs. However, if an event is received late (it has a simulation time earlier than the current simulation time), the simulation is restored (rolled-back) to a time before the event time, possibly sending messages to cancel any erroneously sent event after that time, and restarted. Conservative simulation requires frequent communication, even when no dependencies are present. On the other hand, optimistic simulation can hide the latency of communication by allowing PEs to process speculatively; however, it remains sensitive to communication latency, and incurs the overheads associated with checkpointing and rollbacks.

PDES is difficult to parallelize effectively because of its fine-grained communication behavior and the complex underlying dependency pattern present in most models. Researchers have explored reducing the impact of message latency in a number of ways [14, 61, 67, 46]. Model partitioning [41] and dynamic object migration [57] attempt to localize important dependencies, reducing the frequency of remote communication. Throttling attempts to avoid excessive rollbacks by limiting the simulation from speculating aggressively [62]. However, PDES remains highly constrained by the high cost of communication.

The emergence of multi-core architectures and their expected evolution into manycores presents an exciting opportunity to PDES and similar fine-grained applications. The low

communication latency and tight memory integration among the cores on a multi-core chip substantially reduce the communication cost and have significant impact on the scalability of PDES simulations. However, most existing PDES simulation kernels have been created for cluster environments and have not been optimized to work in multi-core settings, either on individual machines or on their clusters.

In this report, we summarize the results of a two-year project funded by AFRL, with the goal of designing and optimizing PDES to execute in a fast, scalable and reliable manner on multicore and manycore machines and also their clusters.

Our first step towards achieving this goal was optimizing a PDES simulation kernel, the Rensselaer's Optimistic Simulation System (ROSS) [9], for multi-core platforms. Specifically, we re-implemented the process-based simulator as a multi-threaded simulator, to take advantage of the tight integration among cores on the same chip. This allows us to substantially reduce communication latency by passing events directly from one thread to another. In our initial study, we profile the performance of the developed multithreaded ROSS on two multicore platforms: an Intel Core i7, and an AMD Magny-Cours 48-core machine.

We discover a number of performance bottlenecks, especially on the 48-core machine, and propose optimizations to improve their performance. First, we showed that the MPI barrier synchronization does not scale due to lock contention, and use the optimized `pthread_barrier` implementation instead. Second, we show that the standard implementation of memory allocation is not aware of the non-uniform memory latency present on some multi-core architectures and develop message allocation strategies that are aware of these effects. Finally, we show that there is substantial contention for the incoming event queues, and present a distributed implementation that significantly reduces this contention. Together, with these optimizations, the multi-threaded ROSS outperforms the baseline distribution of ROSS by up to a factor of 3 on the Intel Core i7 and 1.2 on 48-core AMD Opteron Magny-Cours.

Second, we examined the performance of this newly developed multithreaded ROSS simulator on the cluster of multicore machines. Indeed, the advantages of multithreaded simulator within a single computing node are limited to small scales: for large scale PDES applications that require more cores than is available on a single node, it is necessary to use a cluster of multi-cores (CMs). In such an environment, the communication delays and software

overheads for communication across machines (inter-node communication) can be substantially higher than those between cores on the same machine (intra-node communication).

Specifically, the question that we address is the following: In the presence of heterogeneous delays, do fine-grained applications such as PDES benefit from the low latency available between some cores, or are they limited by the performance of the slowest links? To explore this question, we perform experiments on a cluster of multi-core nodes connected using Gigabit Ethernet and using MPI for communication across nodes. For cores on the same node, we explore the use of both MPI as well as more efficient thread-based communication exploiting the shared memory hierarchy between the cores on the same node. We show that the remote communication across nodes plays a critical role in determining the performance of PDES. The message processing delays on the communicating thread becomes a performance bottleneck of PDES. As a result, much of the available processing time is consumed in sending and receiving these events. Moreover, the high communication latency results in many messages arriving late, slowing down the simulation progress even further by causing rollbacks.

Therefore, we argue that the impact of the heterogeneous delays must be considered as a first class design consideration when developing PDES algorithms to run on CMs. We then demonstrate three techniques that significantly reduce the impact of the heterogeneous delays. The first technique we investigate is consolidated message routing between machines. In particular, to reduce the impact of message sending and receiving overheads, we combine messages originating from different cores on one machine to different cores on another machine to amortize the software overhead of processing them. Dedicated communication threads combine the messages on the sending side, without delaying messages. On the receiving side, the communication threads extract the individual messages and route them to the appropriate core using shared memory. We further improve the performance of the receiver communication thread by adjusting the frequency of polling for message arrival; with message consolidation, the number of messages is reduced, allowing us to reduce the frequency of the expensive polling operation. Combined, these two optimizations allow the threaded PDES implementation to achieve a 4.5X improvement in performance compared to the process-based implementation. We also consider model partitioning algorithms that are explicitly aware of the high inter-node communication costs and we show that significant additional performance advantages can be realized if such partitioning is used.

Next, we examine the performance and scalability of both multiprocess and multithreaded ROSS simulator on the Tilera architecture. A 64-core Tilera Tile64 chip [71] utilizes a tiled CPU and cache architecture and employs a two-dimensional mesh network as an interconnection fabric between the cores. Compared to mainstream multicore processors and their clusters, the Tilera architecture has a number of unique features that have direct impact on performance and scalability of PDES (or any other fine-grain parallel application in general). First, a significantly higher degree of core integration allows a larger number of parallel threads to communicate efficiently without leaving the chip boundaries, thus creating potential for better scalability. Second, the Tilera architecture features a more balanced communication-computation infrastructure, where the communication bottlenecks are significantly reduced and computation cycles emerge as a more significant bottleneck. The reasons for this are slower processing cores (which increases the fraction of time spent on computation) and well-optimized mesh interconnection network that promotes both low latency and high throughput communication among the cores. These factors have tremendous implications on an application such as PDES, which was traditionally designed with the goal of hiding long communication latencies.

The starting point of our exploration of PDES behavior on the Tilera platform is the multi-threaded implementation of ROSS simulator [9] that we developed as described above. Multi-threaded PDES directly exploits the presence of shared levels of memory hierarchy on the chip (the shared L2 cache in the case of the Tilera) and eliminates delays due to multiple message copying operations and synchronization delays involved in polling of the queues that are inherent in MPI-based implementations.

We demonstrate that a multithreaded simulator also significantly outperforms the MPI-based version on the Tilera, especially when a number of performance optimizations are introduced. In terms of performance optimizations, we propose to adapt three techniques that we proposed for Intel and AMD processors, such that they exploit the features of the Tilera. We also propose some new optimizations that utilize the APIs available from the Tilera Multicore Components (TMC) library; we provide more details on these optimizations later in the report.

When all optimizations are considered, multithreaded ROSS executing the basic Phold model on 56 cores of the Tilera chip (the maximum number of cores that we could use; the other eight cores are reserved for the OS tasks) achieved a speedup of 27X for Phold benchmark

at 20% remote events. This compares to only about 18X speedup achieved by the MPI implementation of ROSS. Next, we study the individual impact of the proposed performance optimizations, both for conservative and optimistic simulation. Finally, we address the issues of object placement and model partitioning in the context of the Tilera TilePro64 platform. Our results demonstrate that while Tilera's mesh network exhibits non-uniform core-to-core latencies, the degree of non- uniformity is minimal. Compounded by the fact that the balanced nature of Tilera architecture makes the applications running on it more tolerant to communication delays to begin with, the minimal non-uniformity in latencies make the PDES performance almost insensitive to the placement strategies (i.e. whether the frequently communicating objects are placed on the nearby or on the distant cores). Furthermore, we demonstrate that the model partitioning strategies that just balance the computational load among the cores (these partitions are much easier to derive), very closely approach the performance of partitioning schemes that try to optimize the number of remote communications (through communication graph mincut), as well as balance the workload. These results are important in that they demonstrate that PDES can exhibit great scalability on the Tilera platform with minimum investment in object placement and partitioning decisions.

Our next effort was to study model partitioning strategies for PDES executing on multicores and clusters of multicores. Partitioning is one of the primary approaches to reducing the impact of communication: by mapping often communicating objects to the same processor or to nearby processors, communication frequency and distance is reduced. Another goal of partitioning is to maintain a balanced workload across the different simulation processes. For partitioning purposes, the simulation model is represented as a graph, where every vertex is a simulation object and every edge represents the fact that the two objects communicate during the simulation. A graph partitioning tool is then used to partition the graph to minimize the cut size (to reduce communication) while maintaining balanced partition sizes (to maintain load balancing) [42, 39].

With few exceptions, partitioning research has focused on graph based partitioning where all objects and edges are considered identical. The advantage to this approach, which we call static partitioning, is it requires only information about the static simulation topology. In practice, this topology often does not reflect the dynamic behavior of the simulation model. In particular, some edges in the communication graph may be significantly more important than

others because the connected objects communicate frequently, or because the events are more critical (little lookahead available between the event generation and its consumption); minimizing the impact of remote communication requires taking this information into account. Moreover, some simulation objects require more computational resources, such as processing and memory, than others either because they become active more frequently or because processing their events requires more computation; effective load balancing must take this information into account. Finally, at a higher semantic level, it is likely that dependency patterns play a role in determining effective partitioning.

We argue that taking the dynamic model behavior into account is critical to effective partitioning of simulation models. We motivate the need for incorporating this information by providing the evidence from real models that both edges and objects have substantially different behavior. Thus, given this information, partitioning can much better localize the most important dependencies, and load balance in a way that takes into account the behavior of objects, rather than just their counts.

We study the partitioning strategies on two multi-core architectures: a dual quad-core Intel Xeon cluster, and a 48-core AMD Magny-Cours system. The two platforms differ significantly in a number of ways, including the relative cost of communication to computation and the behavior of the memory subsystem. We discover that dynamic behavior based partitioning can outperform static partitioning by a factor of 4x on the cluster, and up to 2x on the Magny-Cours system. The best performance is achieved by the partitioning strategy that emphasizes both communication cost and load balancing; this strategy consistently leads to effective partitions for different model types and simulation platforms.

Finally, we study techniques to achieve interference-resilient PDES execution. PDES simulators have traditionally been designed under the assumption of a homogeneous environment with no interference from other co-located applications. Interference from other applications as well as other noise in the system creates competition for the available resources leading to potential slowdowns in parallel applications [52, 77, 72, 59]. In the presence of interference, we expect an application to slow down proportionately to the reduction in its share of the resources: a metric introduced in this paper which we call proportional slowdown. Surprisingly, we found the impact of interference on PDES to far exceed proportional slowdown,

even when the amount of interference is small. For example, when evaluating a multi-threaded fixed-mapping PDES engine, we discover that even 1 external load can result in a performance slowdown of a factor of up to 3.9 for an 8-way simulation on the Core i7 platform, and up to 2.8 for a 48-way simulation on an AMD Magny-Cours platform.

A primary reason for the disproportionately high cost of interference is related to the granularity of the operating system (OS) scheduler. In particular, when the OS schedules an interfering process on a core, it has to context switch one of the simulation threads out, making it inactive. As a result, this thread is stalled, while, assuming optimistic simulation, other simulation processes surge forward. Eventually, when the OS schedules the process again, its late events cause rollbacks throughout the simulation: thus, most of the time on all processes is lost, and additional inefficiency results from the overhead of rollbacks. The problem continues to occur whenever the noise process is scheduled.

We explore interference-resilient execution of PDES on two multi-core platforms: a quad-core Intel Core i7 system, and a 48-core AMD Opteron Magny-Cours platform with Non-Uniform Memory Access (NUMA) latencies. We first propose a dynamic-mapping (DM) scheme that is capable of dynamically changing the mapping between PEs and threads during the simulation. In particular, each thread attempts to work on a PE in a round-robin fashion. For correctness, each PE can only be mapped to one thread at a time. As a result, DM has limited opportunities to solve the problem: a thread is often switched out while in the middle of processing events on a PE. Other threads thus cannot assist and execute the PE until the thread gets scheduled again and releases the lock, causing the PE to lag far behind of others. Thus, although some performance benefit can be obtained, we discovered that the baseline DM cannot effectively solve the interference problem.

To address this problem, we propose an adaptive DM scheme that reduces the number of active threads when interference is detected. As a result, the number of threads is again matched to the available hardware contexts, and the simulation does not have to suffer extended periods when one of its threads is switched out. In this context, the remaining threads have to service a number of PEs that is larger than them. Having the threads switch in round robin fashion among the PEs, promotes load balanced operation but leads to poor locality as PEs move among threads causing cache interrogation. More precisely, the LADM scheme creates a schedule where each

thread is primarily associated with one PE, but spends a portion of its time helping one other PE. The proportion of time is chosen so that the total active time each PE receives remains balanced, avoiding straggler objects. Since each thread works on a limited number of PEs (two under reasonable interference conditions), locality is kept high.

We believe that these directions are highly promising in terms of impacting the design of Parallel Discrete Event Simulators for current and future multicore and manycore architectures. We also have a number of concrete ideas for future work. This report examines the accomplished work in detail and discusses important follow-up directions that can be pursued in future projects.

3.0 Methods, Assumptions and Procedures

In this section, we describe the details of the proposed methods and procedures to improve PDES performance, scalability and resilience to node failures and intervening noise from other processes. We also describe our assumptions about the underlying systems and models.

3.1 Design of a Multithreaded ROSS Simulator (ROSS-MT)

In this section, we first briefly describe parallel discrete event simulation and the ROSS simulator [9], and overview a typical multi-core cluster organization.

3.1.1 Optimistic Parallel Discrete Event Simulation

A parallel discrete-event simulator (PDES) is organized as a collection of Processing Elements (PEs) that communicate by exchanging time-stamped event messages [25, 35]. Each PE processes its events in time stamp order (to ensure causality). PDES simulators differ in the synchronization algorithm used to ensure correct event ordering among events on different PEs. The PEs in conservative simulators exchange messages to upgrade each other of their progress and guarantee correctness. Alternatively, optimistic simulation may be used where PEs process events with no explicit synchronization occurring among them. Causality is preserved among different processes by exchanging time-stamped event messages and using rollback upon receiving a message with a time in the past. Thus, the state of the simulation must be saved to allow rollbacks when a causality breach is detected.

The progress of the simulation (the Global Virtual Time, or GVT) is computed as the minimum of the timestamps of all PEs as well as messages in transit. GVT is used to garbage collect state information, commit output events and often to adapt configuration parameters of the simulation. When a rollback occurs, the state of the simulation is restored to a valid state before the rollback time. Any messages erroneously sent to other PEs must be cancelled by sending anti-messages.

Cascading rollbacks can occur when a rollback at one node causes a sequence of rollbacks at other nodes as it sends out its anti-messages. Cascading rollbacks significantly harm performance. More frequent communication, or higher communication latencies can lead to rollbacks and cascading rollbacks and delays in computing GVT resulting in larger memory footprint, and slower over- all execution and so communication frequency and latency play a

major role in determining the performance of simulation [14].

In our experiments, we use the ROSS [9] simulator. ROSS is an optimistically synchronized simulator. It has the option of implementing reverse computation where, instead of storing state information, code is stored to undo events in case of rollbacks. If no reverse computation code is provided, ROSS uses state saving instead.

3.1.2 Multi-core Architectures

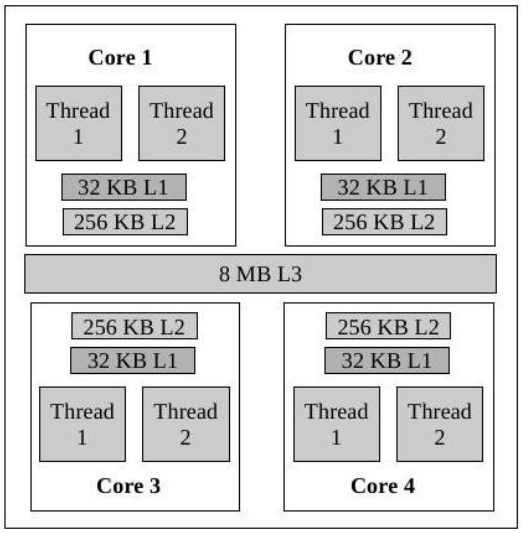


Figure 1 - Architecture of the Intel Core-i7

In our experiments we use two multicore platforms with significantly different architecture and memory organizations. The first is a 4-core Intel Core i7 processor (Figure 1). Each core supports two Simultaneous Multi-threaded (SMT) thread contexts. The cores have private L1 and L2 caches but share an L3 cache. The second platform we use is a 48-core AMD Magny-Cours machine [17]. As shown in Figure 2, there are four CPU chips on the memory bus, each holding 12 cores. The cores on a chip are on two separate dies, with each die holding 6 cores. The cores have private L1 and L2 caches, and share the L3 level of the cache. A specialized interconnect is used to connect the caches across dies. The cores have non-uniform memory access to different regions in memory and experience non-uniform latencies on cache hits to the L3 cache depending on whether the cache line is in the L3 cache of the same die or a remote die.

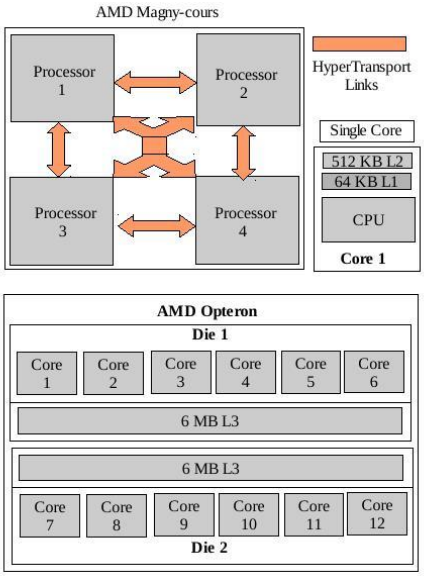


Figure 2 - Architecture of the AMD Magny-Cours

3.1.3 Multi-Threaded ROSS: Design Overview

In this section we overview the components of the simulation kernel that require the use of communication to show the impact of the communication cost on the simulation. We show how communication support is implemented in the baseline MPI-based ROSS simulator. We then overview the baseline threaded implementation of ROSS (i.e. ROSS-MT).

ROSS Simulation Loop

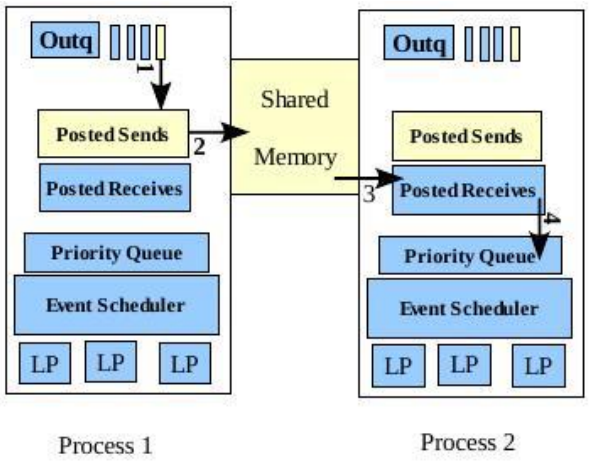


Figure 3 - MPI-based Message Passing Mechanism

Communication occurs in the ROSS simulator for three primary purposes: (1) exchange of event messages; (2) exchange of anti-messages, cancelling earlier messages sent erroneously;

and (3) for Global Virtual Time (GVT) computation which is used to commit events and garbage collect unneeded state and event checkpoint information. It is essential for communication latency to be low for all three of those functions; otherwise, rollbacks occur more frequently, are more expensive and more difficult to contain, and GVT computation overhead becomes very high.

Event message communication in the MPI version of ROSS works as shown in Figure 3. Each PE maintains a queue of outgoing remote events. When a PE sends a message to another remote PE, an event message is first queued in to the Output Queue (Outq). Events are later dequeued from Outq and sent to appropriate destination process asynchronously based on receiver buffer availability. Posted sends and Posted receives buffers are used for asynchronous message passing. Once the event message is successfully received at the destination process, it is queued in to priority queue at the receiver side, while the sender marks the message as successfully sent. The event queue is a priority queue maintained by the scheduler to keep the events in time-order. The scheduler dequeues events from the priority queue and processes them one by one. Due to the need to compute GVT, the state of messages in transit must be tracked. Keeping track of message state enables the appropriate steps to be taken during a rollback as well.

ROSS-MT

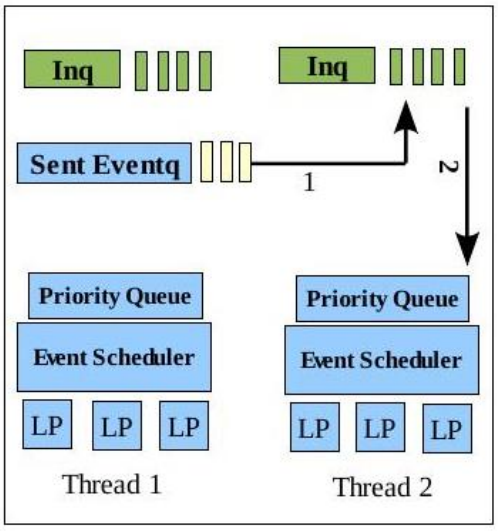


Figure 4 - Multithreaded ROSS Message Passing Mechanism

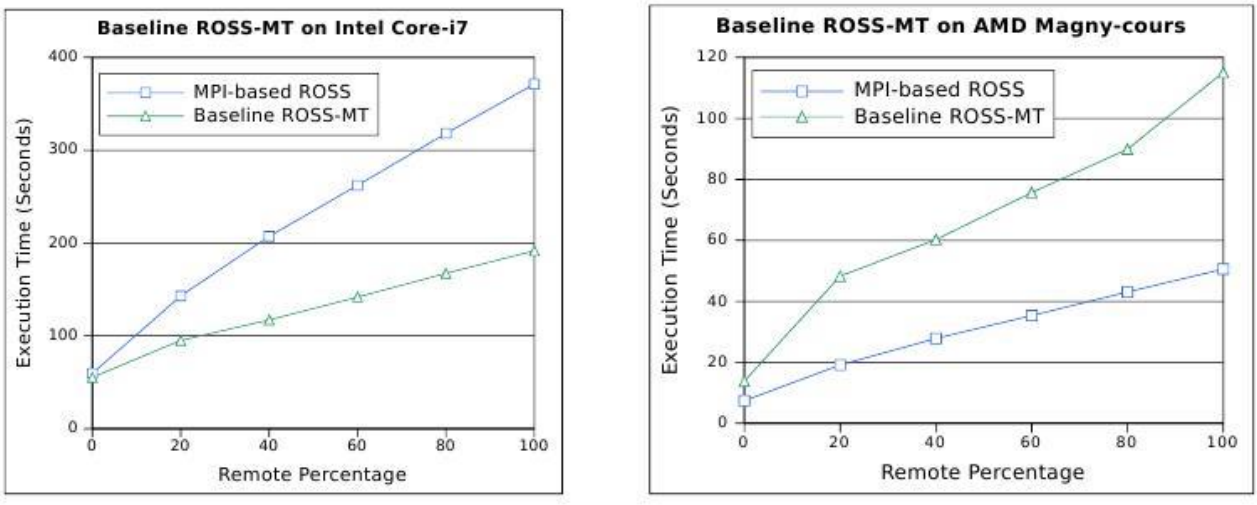
In ROSS-MT we use threads instead of processes, as seen in Figure 4. Because the

threads share the same memory image, there is no need to use explicit message passing between them. Thus, instead of using a separate input and output queue for each thread, we only use an input queue for each thread containing all remote events from other threads (PEs). No buffering is needed and thus Posted send and Posted receive buffers are eliminated. A thread is associated with its own memory manager, scheduler and free event queue for fossil collection.

Communication occurs by inserting a pointer of the message copy in the input event queue of the destination thread. The sender keeps a copy of each message sent so that in case of rollbacks, cancellation messages can be sent. The receiver thread dequeues events from the input queue and inserts them into the event priority queue for processing. Thus we completely avoid synchronization delays present in MPI based ROSS implementation. We use this two stage insertion to avoid lock contention on the main event queue.

3.1.4 Performance Bottlenecks and Optimizations

Figure 5 shows the performance of the basic multithreaded implementation in comparison to the MPI implementation for both the Intel Core i7 (Figure 5(a)) and the AMD Magny-Cours (Figure 5(b)) platforms. For these results, we used the clustered Phold benchmark, which allows us to control the percentage of event messages that are remote; Clustered Phold is described in more detail in the next section. While the Core i7 results show substantial performance improvements with multi-threading, surprisingly, the Magny-Cours results show significant slowdown.



(a) Core i7 (b) Magny-Cours

Figure 5 - Performance of Baseline ROSS-MT vs. ROSS using MPI

The two machines have substantially different architectures, especially with respect to the memory organization. Moreover, the Magny-Cours machine has substantially higher parallelism (48-cores) than the Core i7 (4 cores/8 threads). We profiled the ROSS-MT execution behavior, which allowed us to identify a number of bottlenecks. In this section, we describe three of these bottlenecks and describe optimizations to address them.

Efficient Barrier Synchronization

Barrier synchronization and all-reduce communication primitives are key operations for GVT computation. ROSS-MT uses its own library for barrier synchronization and all-reduce operation. In the baseline version of multithreaded implementation we used condition variables and pthread_mutex for implementing these operations. Profiling results showed very high overheads due to the use of condition variables. We optimized this library by using pthread_barrier construct instead of condition variables.

NUMA-aware Free Memory Management

ROSS implements its own free memory management to avoid unnecessary use of the memory allocation library. The ROSS implementation returns the memory of an event message after it is consumed to a free memory pool. This memory is then used for future message events. Suppose that a message is generated from PE 1 to PE 2. The message is allocated by PE 1 from its closest memory region (the OS NUMA option enforces that). Once the message is consumed by PE 2 it is returned to the memory pool for PE 2. In the future, if PE 2 needs to send an event to another PE, say PE 3, it picks the memory region that was allocated by PE 1, which is remote for both PE 2 and PE 3, leading to high access latency.

To address this issue, we split the free memory pool to keep track of the allocation source. When PE 2 needs memory space for an event, it uses the free memory pool for the receiving PE to ensure NUMA friendly behavior. In addition, we implemented a Last In First Out (LIFO) approach to message allocation. Thus, the most recently freed message is used from each free memory sub-pool. This policy improves cache reuse.

Distributed Locking for the Input Queue

By allowing sending threads to directly access the input queue of receiving threads, we eliminate the need for a buffer copy to an intermediate message queue. However, each input queue may now be accessed by any of the sending threads, as well as the receiving thread (i.e., all threads in the simulation). This gives rise to high contention on the lock to access the input queue. To reduce this contention, we split the input queue into private queues, one for each possible sender. The contention for the queue is reduced from all threads, to only two threads, the sender and the receiver.

We note that the current implementation exploits shared memory to optimize only the message communication aspects of the simulator. There are additional opportunities for optimization that arise due to direct access to other thread's space that we plan to implement in the future. For example, direct cancellation can be used to optimize rollbacks by removing unexecuted erroneous messages directly (instead of using anti-messages) [19]. In the future, we will explore mechanisms to share a single copy of the message instead of creating a copy for rollback purposes.

3.2 Evaluating and Optimizing ROSS-MT for Clusters of Multicores

Our next goal was to evaluate the performance of ROSS-MT on a cluster of multicore machines and come up with optimizations that allow continued improvements in scalability and performance despite the presence of long-latency communication links across the cluster nodes.

We first use an MPI-based Ping-Pong benchmark to evaluate the communication latency in this environment. There are three types of communication in such CMs: intra-core, inter-core, and inter-node, as shown in Table 1. The intra-core communication occurs between two hardware threads on the same core, while inter-core communication happens among different cores on the same node. Inter-node communication is the communication over the network. Table 1 shows these three types of communication latency under different message sizes on the multi-core cluster connected through a Gigabit Ethernet switch. At any message size, the latency of inter-node communication is approximately two orders of magnitude larger than that of other two types of intra-node communication.

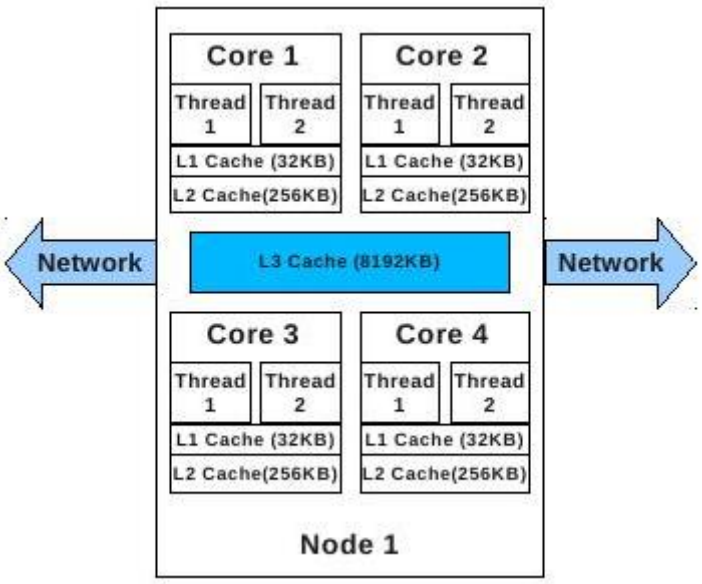


Figure 6 - A cluster of Intel Core-i7 nodes

3.2.1 PDES Performance on CMs

In both types of PDES simulation (conservative and optimistic), communication latency and software overheads play a critical role in determining performance. These overheads determine how fast event messages are communicated affecting simulation progress. The high latency also influences synchronization which has substantial effect on the progress rate and efficiency of the simulation.

Table 1 - Heterogeneous Latency on the Cluster of Core-i7 Nodes (µSecs)

Message Size (Bytes)	intra-core	inter-core	inter-node
4	0.22	0.24	62.38
32	0.25	0.25	62.23
256	0.28	0.3	64.82
1024	0.35	0.38	78.06
8192	1.17	1.42	150.63
16K	2.37	2.88	268.61

Table 2 - Effect of Heterogeneous Latency for baseline ROSS-CMT

	0% remote	20% remote	40% remote	60% remote	80% remote	100% remote
0% regional	7.3 sec	44.7 sec	70.3 sec	95.6 sec	117.5 sec	141 sec
20% regional	10.9 sec	44.4 sec	70.7 sec	95 sec	117.5 sec	N/A
40% regional	12.3 sec	44.9 sec	71.2 sec	95.4 sec	N/A	N/A
60% regional	12.8 sec	45.5 sec	71.9 sec	N/A	N/A	N/A
80% regional	14.3 sec	47.3 sec	N/A	N/A	N/A	N/A
100% regional	15.3 sec	N/A	N/A	N/A	N/A	N/A

ROSS-MT simulator described in the previous section is a step in reducing the communication is limited to a single node. To reach higher scales, we developed an extended version of ROSS-MT, called ROSS-CMT. In ROSS-CMT, in order to avoid the overhead when multiple threads invoke MPI functions simultaneously, only one communication thread on each node performs communication across the network. The communication thread looks up the output queue of each thread in a round robin fashion, and sends remote events to the corresponding destination nodes. Once the communication thread at the receiver side probes (or polls) the event successfully, it then inserts the pointer of this event to the input queue of the destination thread. Generally speaking, ROSS-CMT performs better than ROSS-MPI on CMs; however, the delay of message processing at the side of communication thread imposes a performance bottleneck. We use ROSS-CMT to study the impact of the heterogeneous latencies on CMs.

To characterize the impact of CMs on ROSS-CMT, we use the classical Phold benchmark [26]. Phold is a standard benchmark used in performance evaluation of PDES. It consists of a number of simulation objects distributed among multiple PEs. In our experiments, each PE is mapped to one thread in ROSS-CMT, or one process in ROSS-MPI. During execution each object randomly picks up a target and sends a time-stamped event message to the target. Upon receipt of the event, a new event may be generated to another target. Phold is controllable; allowing the percentage of communication between different objects to be specified to control the ratio between local, regional and remote events. Phold is a synthetic model with simple dependencies among objects, which allows us to study behavior under controlled conditions.

Table 3 - The Rollback Percentage caused by remote events

	0% remote	20% remote	40% remote	60% remote	80% remote	100% remote
0% regional	N/A	100%	100%	100%	100%	100%
20% regional	0%	72.5%	82.3%	86%	87.3%	N/A N/A
40% regional	0%	58.5%	70%	75.5%	N/A N/A	N/A N/A
60% regional	0%	48.9%	60.6%	N/A N/A	N/A N/A	N/A
80% regional	0%	41.9%	N/A N/A	N/A		
100% regional	0%	N/A				

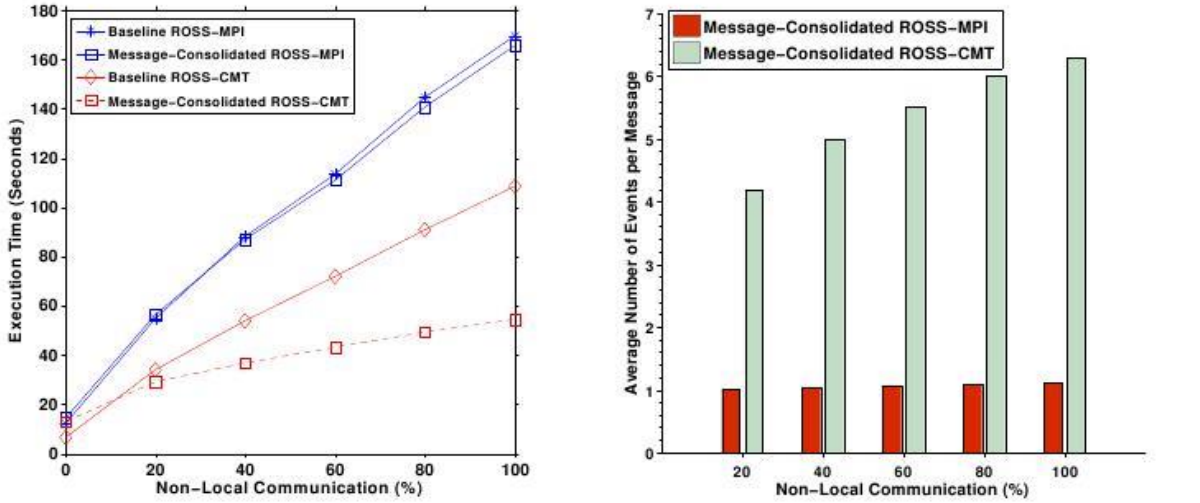
Table 2 shows the performance of ROSS-CMT with optimistic simulation, under different percentages of regional and remote communication on 4 nodes, with 8 threads each. Remote communication refers to traffic across nodes, while regional communication indicates the communication between cores on the same node. The rest of the communication is local

and occurs within the same PE. Table 2 shows that the execution time increases substantially as the remote communication increases; at the case of 80% remote communication and 20% regional communication, the simulation runs approximately 10 times slower than 0% remote communication case with the same percentage of regional communication. Clearly, the impact of regional communication for ROSS-CMT is much less than that of remote communication. For example, at 0% remote communication, the performance drops only from 7.3 seconds to 15.3 seconds when regional communication increases from 0% to 100%.

Table 3 shows the percentage of rollbacks caused by remote communication. At the case of 20% remote communication and 20% regional communication, 72.5% of total rollbacks are caused by remote events. Rollbacks occur when events are arriving late and the local simulation time proceeds beyond their simulation time. It is considerably more likely for a rollback to be triggered by an incoming remote event, rather than regional event, because of the high latency on the slow communication link used to send the remote events. Thus, it is necessary to consider optimizations to reduce the cost of remote communication in PDES.

3.2.2 Managing Heterogeneous Communication Latency

In this section, we discuss the use of three optimizations to reduce the message communication overheads, and to hide the inter-node communication latency on CMs. The theme of these optimizations is to focus on the impact of the expensive communication links, by reducing the frequency of communication across them. We first describe the implementation of message consolidation on ROSS-CMT, where multiple messages are combined and routed through the slow links together. Next, we build up on the message consolidation technique by exploiting the observation that fewer messages now arrive from the distant links. To capitalize on this, we propose infrequent polling to reduce the frequency of the expensive operation to check for the incoming messages. Finally, we investigate making the high cost links visible to model partitioning in ROSS-CMT to better map the model around them.

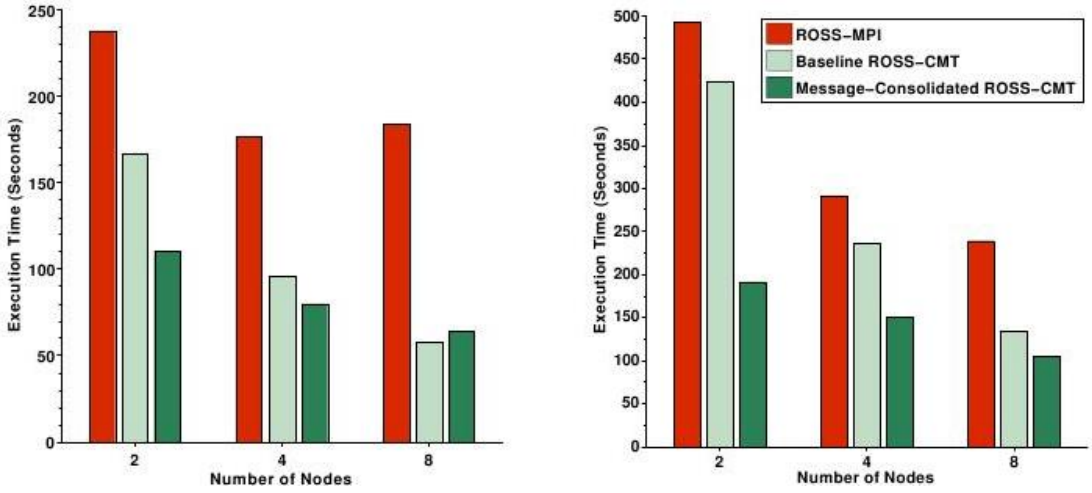


(a) Execution Time Comparison (b) Average Number of Events per Consolidated Message

Figure 7- Performance of Message Consolidation on 32-Way PDES Simulation

Optimization 1: Message Consolidation and Routing

Message send operations across nodes incur significant overheads including multiple buffer copies and system calls/OS delays on both the sender and receiver sides. Therefore, when inter-node communication is frequent, these overheads can dominate, increasing the message processing latency, but also potentially delaying critical messages and slowing down overall application progress.



(a) 20% Remote Communication (b) 80% Remote Communication

Figure 8 - Impact of Message Consolidation on PDES Simulation for Different Number of Nodes

We address this limitation by employing an optimization called message consolidation and routing. Message consolidation creates designated communication threads on each node that act as consolidation points for communication. Instead of communicating directly, threads prepare their outgoing messages which are then collected by the communication threads and consolidated when possible to reduce communication overhead. Messages sharing the same destination node (regardless of the destination thread) can be consolidated at the sender side. At the receiver side, the messages are deconsolidated and delivered to the appropriate thread.

This approach bears some similarity to traditional message consolidation (also known as message aggregation) [14]. In these approaches, messages from the same sender to the same receiver are consolidated to amortize overhead. Often, the sender artificially delays messages in hope of receiving additional messages to send to increase the opportunity for aggregation. In contrast, the proposed optimization combines messages from different senders to different receivers as long as these messages share the source and destination nodes. As such, it exposes significantly higher opportunities for consolidation and avoids the need for delaying messages in hopes of receiving later messages for consolidation. In other words, consolidation is not only carried out over time, it is also carried out across different senders and receivers sharing the same source and destination nodes. By focusing consolidation on the slow inter-node links – no consolidation is carried out between cores on the same node – we achieve the highest reduction in communication overheads.

A critical parameter for message consolidation is the number of messages consolidated in one send. In our approach, we set a threshold based on the cumulative size of the consolidated message. This approach allows us to match the sent message size to the underlying communication medium maximum payload size in order to avoid expensive MAC layer fragmentation. For Ethernet, the maximum payload size is 1500 bytes, which allows us to aggregate up to 10 event messages in ROSS-CMT. The use of Ethernet jumbo frames could offer room for higher degrees of consolidation, especially for applications that have large size messages.

Optimization 2: Infrequent Polling for Incoming Messages

In order to detect the incoming remote messages, ROSS probes (or polls) the network after every event. However, probing is an expensive operation, and probing too aggressively increases overhead, often discovering that no message is available. If we adjust the polling

frequency, un-necessary expensive probes can be avoided. The probing frequency can be adjusted based on the behavior of previous probes to reach an effective probing rate that balances the overhead of probing against the loss of efficiency that may result if some messages are received late. We also note that message consolidation can benefit probing because it reduces the message frequency, allowing us to probe less frequently.

Infrequent polling has been proposed before for optimizing the performance of asynchronous applications such as PDES [67]. However, when applying message consolidation in a CM environment the behavior is significantly different because a single thread polls for a group of simulation threads, making the communication pattern different. The interplay between message consolidation (which reduces the number of overall messages) and infrequent polling has not been studied before.

Optimization 3: Exposing Heterogeneous Latencies to Model Partitioning

Partitioning can play a significant role in reducing the communication overhead for parallel applications [66, 41]; by keeping the most heavily communicating objects together, the overhead of communication can be controlled. A joint consideration of partitioning is maintaining load balancing between the processing elements by evenly distributing the work among them.

In an environment with heterogeneous delays, the higher delays between nodes can be exposed to the partitioning tool to allow it to make more informed partitioning decisions. Without this information, the work would be simply partitioned between the cores without consideration to the heterogeneous delays between them.

Since there is no static communication structure in the Phold model, we use the hierarchical Phold model introduced in [2], to study the impact of partitioning on CMs. In this model, groups of objects are arranged in a hierarchical communication structure. Object groups closer to each other communicate more often while the farther groups have progressively less communication. Although this is still a synthetic model, it exhibits features of real models both in terms of topology and communication pattern. We use a partitioning algorithm that profiles the simulation model and uses its behavior information to carry out partitioning [2]. The implementation uses a state of the art partitioning engine (hMetis [36]) to partition the

simulation graph which is annotated with the profiling information. In our experiment, two types of partitioning strategies are compared. A latency heterogeneity-sensitive partitioning computes a node-level partition first and then each part is partitioned again at the core level. This strategy ensures that the groups of objects with most communication will be placed on the same node. It is compared with a latency-oblivious strategy where the model is partitioned across cores, without differentiation between the heterogeneous latency among them.

3.3 Evaluating PDES on the Tilera Architecture

Our next project goal was to evaluate and optimize the performance of PDES on the Tilera platform that was supplied for this purpose by AFRL.

3.3.1 Background on the Tilera Architecture

TilePro64 is a power-efficient 64-core processor from Tilera. It uses switched, on-chip mesh interconnect providing coherent dynamic distributed cache. The processor chip is comprised of 64 power efficient cores (tiles) arranged in the form of an 8x8 matrix. Tiles are connected by six mesh networks forming tight integration of cores. The cache coherence across the cores and the memory provides efficient and scalable platform for shared memory applications. The role of the mesh network is to move data between cores, memory and I/O providing low latency and high bandwidth.

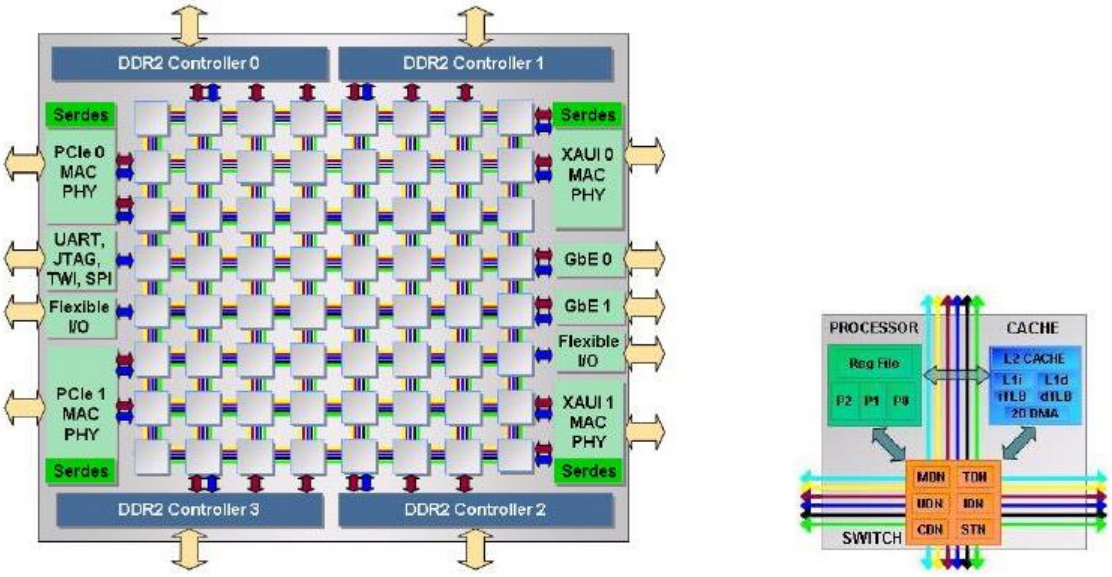


Figure 9 - Architecture of the Tilera Processor (used with permission from Tilera Corporation)

The iMesh Interconnect consists of two classes of networks: the first class comprises a set of software-visible networks for application-level streaming and messaging, while the second consists of the networks used by the memory system to handle memory requests, exchange cache coherency commands and support high performance shared memory communication. Dedicated Switch Engines are used to implement the iMesh Interconnect, allowing for a complete decoupling of data routing from the Processing Engines. The Switch Engine contains six physical mesh networks. The Static network (STN) switches scalar data between tiles with very low latency. The other five are dynamic networks, which facilitate streaming and packet data transfer among tiles and I/O devices. Of the five dynamic networks, namely the UDN, TDN, MDN, CDN and IDN, only the User Dynamic Network (UDN) is visible to the user. The others are used to satisfy cache misses from external memory and other tiles, for direct memory access (DMA) transfers, for I/O, and for various other system-related functions.

A single processing tile has a 32-bit 5-stage very long instruction word (VLIW) pipeline with L1 instruction and data caches, L2 combined data and instruction cache, and a routing engine for the mesh networks. The 64KB L2 caches from each of the cores form a distributed L3 cache accessible by any core and I/O device. Static branch prediction and in-order execution further reduce area and power required. Translation look-aside buffers are present on each core and support memory protection for virtual memory. Each memory controller reorders memory read and write operations to the DIMMs to optimize memory utilization. Cache coherence is maintained by each cache-line having a home core. Upon a miss in its local L2 cache, a core needing that cache-line goes to the home cores L2 cache to read the cache-line into its local L2 cache. Two dedicated mesh networks manage the movements of data and coherence traffic in order to speed the cache coherence communication across the chip. To enable cache coherence, the home core also maintains a directory of cores sharing the cache line, removing the need for power hungry bus-snooping cache coherency protocols. Because the L3 cache leverages the L2 cache at each core, it is extremely power efficient while providing additional cache resources. Figure 9 shows the I/O devices, 10G and 1GB Ethernet, and PCI-e, connecting to the edge of the mesh network. This allows direct writing of received packets into on-chip caches for processing and vice-versa for sending. We believe this feature can be exploited by PDES in a clustered environment.

The Tilera platform provides the iLib library which allows parallel programming and provides APIs similar to MPI for message send-receive, all reduce operation and barrier synchronization primitive. We use MPI library implementation (provided by ISI from University of California Santa Barbara) which acts as a wrapper for iLib APIs and makes MPI application portable on Tilera platforms. iLib internally uses User Data Network (which is iMesh) and provides buffering mechanism for message send-receive.

3.3.2 Adaptation of ROSS-MT Performance Optimizations to the Tilera

The first optimization targets efficient cache usage for free memory management. ROSS implements its own free memory management to avoid unnecessary use of the memory allocation library. We introduced the LIFO approach to message allocation from the free queues. In this scheme, the most recently freed message is used from each free memory sub-pool. This policy improves cache performance and significantly reduces the number of cache misses. For the Tilera platform, we also added a new optimization by enabling a thread-specific heap feature available in the Tilera (TMC) library to enhance the local cache usage.

The second optimization discussed previously, is the distributed locking for the input queue. We observed that on the traditional multicore platforms, lock contention among the threads for the shared input queue becomes a significant bottleneck. To reduce this contention, the input queue is split into multiple input queues and a group of senders share an input queue. However, maintaining too many queues increases the overhead needed to poll them. Therefore, there is a trade-off between the lock contention overhead (in case of too many threads sharing a queue) and queue polling overhead. Our study of ROSS-MT for traditional platforms showed that due to the much higher impact of the lock contention in traditional Intel and AMD multicore systems, the optimal performance was achieved when one queue was used for each sender and receiver (meaning that queue polling overhead was relatively low on those systems).

In contrast, on Tilera we observed that the lock contention is a much lesser issue due to efficient inter-core communication network and that the queue polling overhead (which requires the extra core cycles) is dominant. Therefore, the optimal number of senders sharing a queue needs to be reconsidered, if this optimization is used on Tilera. Specifically, our experiments demonstrate that a single input queue can be shared by eight senders and one receiver without

experiencing any lock contention. In order to further reduce lock-unlock overhead on the Tilera, we use the Spin_queued_mutex primitive supported by the Tilera (TMC) library. Spin_queued_mutex are special spin_locks that require a smaller number of cycles compared to pthread_mutex to implement lock and unlock operations.

Finally, the third optimization that we proposed for ROSS-MT targets efficient barrier synchronization. This is important, because barrier synchronization and all-reduce operation are key components for GVT computation – a critical PDES subsystem. ROSS-MT implementation uses its own library for barrier synchronization and all-reduce operation. Our library uses pthread barrier which uses atomic instructions directly supported by the ISA to optimize barrier operation. We observed that barrier synchronization based on condition variables and pthread_mutex has very high overhead at high degree of parallelism.

3.4 Model Partitioning Based on Dynamic Behavior for PDES

Our next goal was to investigate model partitioning schemes for PDES. We first motivate the need for using dynamic information during model partitioning.

Real world simulation models exhibit dynamic activity patterns which static partitioning approaches are unable to exploit. Many phenomena exhibit skew in their behavior often in a way that does not correlate with their structure. In this section, we present two representative examples to demonstrate that such activity patterns exist.

Table 4 - Protein-Protein Interaction

Interaction Score Cut off	Number of Protein-Protein Interactions Above Cut Off Score
0.25	79441
1	37606
2.5	25598
25	5394
250	1232
2500	498

Source: http://www.compbio.dundee.ac.uk/www-pips/dbStats.jsp

3.4.1 Example 1: Protein-Protein Interaction Networks

Systems biology is the study of the functional biological systems observed through the use of both wet lab and dry lab experiments. However, due to cost, most experiments are observed in labs

first and then simulated to acquire further results [22]. One particular area in systems biology that exhibits interesting communication patterns is protein-protein interaction networks. In these networks, the degree of connectivity between proteins follows a power law distribution [38]. Another interesting aspect of protein interaction networks is how likely two connected proteins will interact. An example of the distribution of protein-protein interaction likelihood can be found in Table 4. In this table an interaction score of 2 means that two proteins are twice as likely to interact as arbitrary routine pairs [48]. Clearly, a large skew in interaction can be observed; this impacts both the communication between these objects, as well as the amount of processing they do.

3.4.2 Example 2: P2P Networks

We also analyzed a P2P networking simulation benchmark and observed similar trends. In particular, Table 5 shows percentages of communicating object pairs and percentage of total communication instances when we only consider the object pairs that communicate at least the number of times defined by the communication frequency threshold parameter that is shown in the first column of this table. The results present the average communication frequencies (second row), as well as the communication frequencies observed at various other thresholds (including 10 standard deviations away in the last row). Even at this high threshold value, a significant percentage of all communication events is encountered (16%), while the number of distinct object pairs that contribute to it is very small (less than one tenth of one percent of all communicating object pairs).

The conclusion is that the communication patterns exhibit high skew, and the communication graph has a small number of edges with very high weights (activities) and a much larger number of edges with smaller weights. Clearly, the information about the structure alone is not sufficient to capture these dynamics.

As the two examples above demonstrate, skewed, and even power-law distributed behavior, is quite common in real models. Quite often, this behavior does not match the structure (or is even hidden by it). As a third example, the Internet topology is known to display power-law connectivity (structure) [30]. Commonly held understanding [50], translated into widely used simulation models [49], assumed that the core of the network is where the highest degree nodes existed. However, it was later shown that the edge routers have the highest degrees (to

connect end customers) while core routers had small degrees due to the difficulty of scaling the number of interfaces on high speed routers [40]. Thus, structure would not identify core routers as the most active, potentially partitioning neighboring core routers apart, or failing to account for their disproportionate activity when load balancing.

Table 5 - P2P Network Simulation

Communication Frequency Threshold	Number of Communicating Object Pairs	Percentage of Communicating Object Pairs	Number of Communication Instances	Percentage of Total Comm. Instances
1	37020	100	645436	100
17 (*Avg*)	6396	17	586740	90
131 (*Avg* + *stdev*)	1260	3.4	358556	55
245 (*Avg* + *stdev* * 2)	116	0.3	169224	26
1157 (*Avg* + *stdev* * 10)	26	0.07	100672	16

Traditionally, PDES partitioning has been focused on static partitioning which can exploit the structural properties of the models. However, as we have shown, the activity patterns can differ significantly from the underlying static connectivity. Static partitioning can be ineffective or even harmful for such models. This is the motivation behind the work in this paper.

3.4.3 Dynamic Partitioning based on Model Behavior

We have motivated the need to incorporate dynamic model behavior into partitioning decisions. In this section, we consider the problem of how to implement such a partitioning scheme.

There are two primary challenges that must be addressed:

1. *Extracting dynamic model behavior.* Obtaining dynamic activity information is not straightforward; it requires either profiling the model, static analysis of the model, or hints from the model developers.

2. *Partitioning based on dynamic information*. Once the behavior information is obtained, the second step is to exploit it in partitioning algorithms. Our approach annotates both the edges and the objects of the connectivity graph with weights derived from the dynamic information. The weighted graph is then partitioned. We discuss the approach in more detail in the remainder of this section.

In this study, our focus is on the second problem: once the dynamic model information is available, how do we exploit it to produce better partitioning. We obtain the dynamic model

information through profiling. The reason for focusing only on the partitioning problem is that it allows us to evaluate the size of the available opportunity. Once we establish that dynamic model behavior based partitioning can yield superior performance, our future work will address the first problem to enable practical exploitation of behavior information in partitioning.

Our approach to partitioning is the following. We profile the models to obtain the dynamic communication pattern between the objects, as well as the activity pattern of the objects themselves. This communication information is used to derive weights for the edges in the static connectivity graph. Similarly, the object activity is used to derive weights for the vertices in the same graph. We can now apply partitioning on the weighted graph; by minimizing the weighted mincut, the partitioning tool minimizes the dynamic cutsize (the number of remote messages, rather than remote edges). Similarly, by load balancing the object weights, the run-time is load balanced across PEs (rather than the number of objects across PEs).

To evaluate the importance of dynamic partitioning, and the relative importance of object weights to edge weights, we investigate the following six partitioning strategies:

1. **Random:** Random partitioning does not take into consideration any connectivity or activity information. It places an equal number of arbitrary objects on each processor regardless of their relationships to each other. Random strategy represents a baseline of no partitioning algorithms applied to determine object placement.

2. **Static:** Static scheme partitions a static connectivity graph, using static information for both edges and objects. All edges and objects are treated equally regardless of their varying importance. Static partitioning represents the baseline of existing partitioning approaches.

3. **Object-Only:** Object-only partition strives to balance total weight of all the processors. It does not, however, consider inter-object relationships. Thus, it considers dynamic object weights, but ignores connectivity information. This strategy is expected to perform well when the impact of balanced object workload is much more important than the inter-object communication.

4. **Activity:** This strategy takes into account the dynamic edge activity information, but only the static object weights (e.g., all objects have the same weight). The weight of the

edge indicates the importance of the relationship between two objects, for example, as a function of how often they communicate or the criticality of these events (lookahead available between generation and execution time of the event). In models dominated by communication costs, this strategy performs well.

5. **Object-Activity**: This strategy takes into account both the dynamic object and edge weights. It is expected to provide the best performance across different simulation models as it minimizes communication as well as balances the workloads.

We use the hMetis partitioning package [36] for graph partitioning. hMetis is a state of the art partitioning tool implementing both bi-partitioning and k-way partitioning for general weighted graphs. hMetis works by first creating a mincut partition, and then attempts to exchange objects to satisfy the load balance requirement. As a result, it is not necessary to normalize the edge and object weights to each other since they are not jointly optimized. The load imbalance constraint is specified by providing an imbalance factor to inform the partitioning tool of how much imbalance can be tolerated.

All of the partitioning strategies other than Random and Object-Only use hMetis. For Object-only, we use a simple bin packing heuristic where the next largest weight object is assigned to the processor with the least total weight.

3.4.4 Experimental Methodology for Partitioning Studies

PDES performance evaluations often use synthetic benchmarks, such as PHOLD [25, 55], because of the lack of large scale portable models to enable comparison across simulators and infrastructures. In PHOLD, the PEs are allocated an equal number of objects, and each object is initialized with the same number of events (on average). During simulation, each object randomly picks a target and sends an event to that target. Upon receipt, the target picks another object and sends an event. The total event population is preserved at all times. PHOLD is simple, and is generally effective for testing system performance in a controlled way. Extensions of PHOLD [33] to control remote communication percentage, as well as more general synthetic models [4] have been proposed. However, the behavior remains different from dynamic models and it is difficult to use them to evaluate model-related algorithms and techniques. Other environments [63] appear promising in that they take in the model topology and activity, but are

specific to a simulation environment and were not available to us.

We developed a configurable synthetic benchmark that enables composition of abstract models with various structure and dynamic properties to represent real world models. Importantly the model allows independent specification of topology and dynamic behavior.

Topology defines the static structure of the simulation: which objects communicate to what other objects. Without loss of generality, we implemented two classes of topology: (1) *Hierarchical* models create a tree hierarchy of object groups. Objects have dense connectivity to nearby objects in the tree; there are sparse connections between remote objects. Structurally, this model is similar to systems such as road transport networks where there is dense network of roads inside a city while on a second inter-city level they are connected with a sparse highway network. The model is controllable in the number of levels in the tree, and the connectivity intensity at each level; and (2) *Uniform* model, on the other hand, does not exhibit structural variations in connectivity. It's unclear whether such models exist in practice, but we wanted to be able to evaluate partitioning performance when the structure carries no clustering information (which may help or harm partitioning depending on the dynamic behavior of the model). The model is configurable in the intensity of connectivity.

To represent various communication patterns, we use a Pareto distribution to control activity. The Pareto distribution allows us to controllably vary the activity pattern, creating skew that is independent of the underlying topology. In addition, the Pareto distribution is also optionally applied to object computation requirements to simulate different workload for each object. The distribution can be configured so that at one end nearly uniform distribution is obtained, while at the other, a heavy tailed distribution is obtained where some edges (or objects) have much higher importance than others. The Pareto distribution is described as follows.

$$Frequency = \left[\frac{-(UH^{\propto} - UL^{\propto} - H^{\propto})}{(H^{\propto}L^{\propto})}\right]^{-1/\propto}$$

where U is uniformly distributed on (0, 1) , H is the upper bound while L is lower bound.

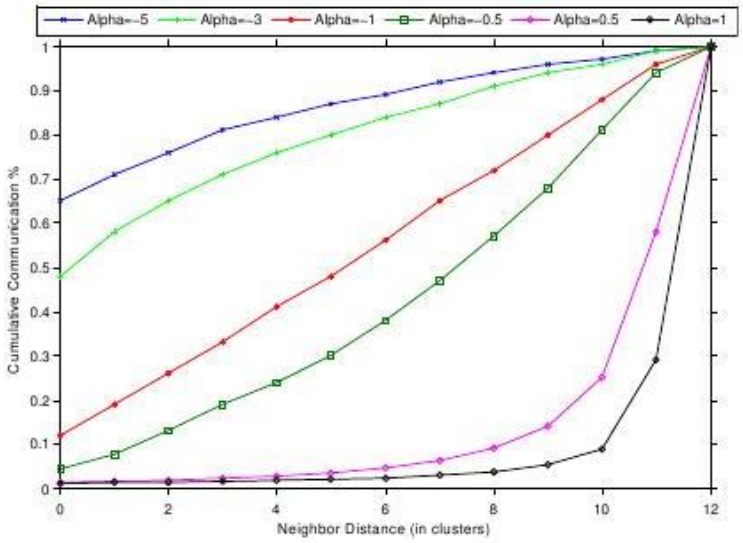


Figure 10 - Communication Frequency Distributions

Figure 10 shows the frequency distributions at 6 values of parameter α. The communication pattern becomes gradually skewed with the increasing value of α. These communication levels however indicate the frequency of messages at each level in the topological model.

To ground the description above, and to provide a description of the simulation model used in the experimental section, we describe how we instantiate a hierarchical model example. A primary parameter in this model is number of *Levels (L)* in the hierarchy. At the leaf level, we start with a predefined number of objects *x*. Every subsequent level consists of two of the clusters below it. Therefore, the total number of objects in this model is x^L. A third parameter controls the number of neighbors each object has at each level. During initialization the object randomly chooses that many neighbors from the cluster at corresponding level and guarantees that the communication will be forwarded to one of these neighbors. As the candidate group size increases at upper levels but the number of edges remains the same, the density of edges is higher at lower level groups while it gradually reduces at higher levels. This creates a hierarchical static connectivity structure.

On top of this topology, we use the Pareto distribution with the specified α parameter to control the frequency of communication at each level. The steeper the curve, the more intensely we skew the communication frequency. For example, if we use α = -5 then most of the communication will be forwarded to the neighbors in nearest cluster. On the other hand, if we

use α = 1, most of the communication will be forwarded to the neighbors in the farther clusters.

3.5 Supporting Resilience to External Interference

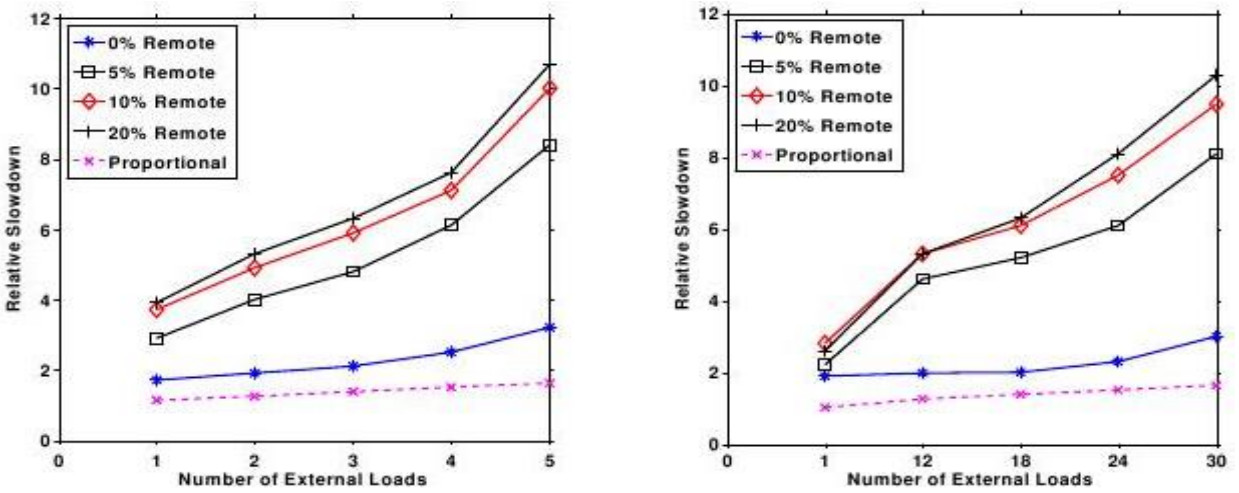
Finally, our last goal for the project was to design PDES in a way that makes it resilient to significant performance slowdowns due to the presence of intervening noise from other processes in the system.

3.5.1 Ideal Slowdown under Interference

Consider a PDES simulation running with N_p threads on a multi-core platform. Let N_c be the total count of hardware threads such that all these threads can execute at the same time; hardware threads refers to cores, or hardware contexts in the case of Simultaneous Multi-Threaded (SMT) processors. Suppose that an external interfering load can start and terminate at any time during the simulation. Thus, to measure performance more accurately, we divide the simulation into *n* small intervals $[X_{j-1}, X_j]$ indexed by j. In addition, let N_{total} be the total number of software threads executing on the machine (i.e., the number of PDES threads, as well as the number of external loads running concurrently) during the interval j. We assume that the operating system scheduler fairly allocates its CPU resources to each thread. In other words, each load obtains $\left(\frac{N_c}{N_{total}}\right)$ of the available CPU time on average during the interval j, assuming that N_{total} loads compete for N_c CPUs. Therefore, the expected PDES slowdown under such conditions during the interval j is approximated by:

$$S_j = \frac{N_{total}}{N_c} = \frac{N_p + N_e}{N_c} \tag{1}$$

where N_e is the number of external loads running concurrently with PDES during the interval j. Note that the above reasoning assumes that threads are computation bound and are therefore available to run whenever the scheduler schedules them. We call S_j the proportional slowdown during the interval j, since S_j increases proportionately to the number of interfering load processes. We assume that N_{total} is always greater than or equal to N_c, and the interference from external loads on PDES performance occur if $N_{total} > N_c$.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 11 - The Relative Slowdown of ROSS-MT caused by External Loads

The run time of the entire PDES simulation in the presence of external loads can be approximated by adding up the expected run time across all intervals. Let T_j be the execution time required for the interval j of a FM simulation without interference. By multiplying T_j by the corresponding S_j, we obtain T'_j, defined as the execution time required for the interval j of the simulation in the presence of external loads. Therefore,

$$T_{ideal} = \sum_{j=1}^{n} T'_j = \sum_{j=1}^{n} T_j \times S_j \quad (2)$$

denotes the ideal runtime of the entire simulation in the presence of external loads. T_{ideal} represents a best case scenario where the presence of interference merely reduces the amount of available resources and results in a slowdown proportional to this reduction. We will show that in practice, the impact is significantly worse than T_{ideal} because of the dependencies between the threads belonging to one application.

3.5.2 Measured Impact of Interference

In the previous subsection, we defined proportional slowdown as a metric that expresses the ideal slowdown of an application in the presence of interference from external processes. In this section, we evaluate the slowdown experienced by both the ROSS-MT and WarpIV PDES simulators, showing that both far exceed proportional slowdown. In the next section, we start

exploring approaches to improve the performance of simulation in the presence of interference.

PDES Slowdown Under Interference

For most of the experiments, we use the Phold simulation model [26], which equally distributes a number of simulation objects among PEs. We use a controllable version of Phold that allows specifying the communication percentage between different objects on different cores. The simulation consists of 8 PEs running on the Intel Core i7 platform, and 48 PEs running on the AMD 48-core machine, with 1000 objects per PE. Each PE was also mapped to a different thread, thus all were used by ROSS-MT threads in the absence of external loads. In addition, we selected a GVT interval of 128 on both platforms, with a batch size of 24 events. Although the results are somewhat sensitive to the GVT interval (as a small GVT interval acts as a throttle to the simulation [69]), these values are in the range where ROSS-MT is most efficient across a range of models.

We use a CPU-intensive process as the external load; the process repeatedly performs computation within a tight loop. Thus, the process when active competes continuously for CPU cycles with the ROSS-MT threads. In this first set of experiments, the external load is started with ROSS-MT, and executes for the duration of the simulation. Thus, *proportional slowdown* from 1 external load can be calculated by Equation 1 to be $\frac{9}{8}$ for the Core i7 and $\frac{49}{48}$ for the AMD Magny-Cours. In these experiments, we do not set the CPU affinity for either ROSS-MT or the noise process, providing the OS scheduler complete freedom in scheduling the processing threads to the hardware resources.

Figure 11(a) and Figure 11(b) show the relative slowdown experienced by ROSS-MT as the number of external loads increases, on the Intel Core i7 system and AMD Magny-Cours machine respectively. The relative slowdown is calculated by dividing the execution time of simulation in the presence of interference by the one without interference. We show these results as the percentage of remote communication is increased, which increases the dependencies among the different PEs. ROSS-MT with 0% remote communication performs close to proportional slowdown: since there are no dependencies between PEs, if a PE is delayed it does not affect the progress at other PEs. The OS scheduler does not always context switch out the same thread; thus, all threads make progress with their computation. In contrast, the interference from external loads dramatically degrades the performance of ROSS-MT even when a small

amount of remote communication exists, far beyond proportional slowdown. For example, even 1 external load can result in a performance slowdown in ROSS-MT of a factor of up to 3.9 on the Intel Core i7 machine, and up to 2.8 on the AMD Magny-Cours machine; these values far exceed the ideal slowdown of 1.125 and 1.02 for the Core i7 and the Magny-Cours respectively.

The problem is not specific to ROSS: we were able to demonstrate similar trends, and even worse slowdown, on the WarpIV PDES simulator [75]. Table 6 show 4-way optimistic and conservative simulations interfered by 1 external load on a quad-core processor; due to export control restrictions on WarpIV, we had to run this experiment on a quad-core Xeon machine. Somewhat surprisingly, the simulation almost stops when the external load takes 100% of the time on one core (a situation which occurred some times, a decision that the Linux scheduler makes). We believe this situation is due to the fuzzy barrier used in GVT computation in WarpIV [32]. At any given time, one thread is not executing and the fuzzy barrier condition is not met. However, even when the external load gets a lower scheduling priority and shares one of the CPU cores with a PDES process, the WarpIV simulation still experiences a performance slowdown of a factor of about 2. The situation was the same for both conservative and optimistic simulation.

Table 6 - Execution Time of a 4-way Simulation on a Quad-core Processor using WarpIV Simulator

	Optimistic	Conservative
No External Load	6 sec	10 sec
1 External Load takes 50% CPU of a core	12 sec	19 sec
1 External Load takes 100% of a core	~ 4.7 hours	~ 40 hours

Explaining the Impact of Interference

Table 7 and Table 8 show the efficiency of ROSS-MT achieved on the two multi-core platforms. The interferences from even one external load substantially reduces the efficiency of the simulation (from around 95% to around 61% on the Intel Core i7 platform). Additional interfering processes further degrade efficiency.

Table 7 - Efficiency of 8-way Simulation on the Intel Core i7 machine

Remote Communication (%)	Number of External Loads					
	0	1	2	3	4	5
0	100%	100%	100%	100%	100%	100%
5	94.6%	61.0%	51.4%	48.0%	45.3%	42.3%
10	96.3%	51.6%	46.0%	42.9%	40.8%	38.3%
20	96.8%	49.8%	43.6%	40.9%	38.8%	36.8%

Table 8 - Efficiency of 48-way Simulation on the AMD Magny-Cours machine

Remote Communication (%)	Number of External Loads					
	0	1	2	3	4	5
0	100%	100%	100%	100%	100%	100%
5	95.9%	78.5%	47.0%	44.5%	45.4%	42.1%
10	96.4%	65.7%	41.7%	39.5%	39.0%	38.0%
20	96.9%	66.5%	41.5%	39.0%	36.6%	35.9%

To understand the drop in efficiency and the resulting slowdown, we first explain the event processing mechanism within ROSS-MT. As is typical with most PDES simulators, each thread is assigned a unit of work comprising of a group of objects (PE). The groups of objects assigned to each thread are selected, often via a partitioning algorithm (e.g., [3]), to minimize costly communication and to load balance computation. Each thread is responsible for processing all events whose destination is an object in its PE group. Thus, the mapping of work to threads is fixed.

Consider a 2-way simulation of ROSS-MT, with 1 LP per PE, as seen in Figure 12. PE 1 and PE 2 are executed by thread 1 and thread 2 respectively. Suppose an external load starts and interferes with thread 2 at wall clock time t_1, after a GVT computation phase (which requires barrier synchronization in ROSS). Once the interfering noise process is scheduled, thread 2 is context switched out and stops execution, while thread 1 continues. Thread 2 does not get scheduled again until the noise process exhausts its OS quantum (or otherwise, some other hardware context becomes available); the OS quantum is typically in the 10s of milliseconds, sufficient for Thread 1 to execute several million CPU cycles. At a wall clock time t_2 ($t_2 > t_1$), thread 2 resumes execution, and PE 2 sends an event e_1 to PE 1. Due to the large pause in execution, this event is most likely a straggler as PE 1 has executed far ahead of PE 2 limited only in the ROSS case by the GVT computation interval; in other simulators, the degree of optimism can be unbounded.

Upon receiving e_1, PE 1 is rolled back to a simulation time before that of e_1, and then is re-executed. Thus, not only is processing time lost at PE 2 while it is context switched out, but most

of the time available to PE 1 is also wasted, which explains why the slowdown exceeds proportional slowdown. The overhead of large rollbacks in terms of state restoration (or reverse computation), sending anti-messages, and other data structure restoration exacerbates the inefficiency. This effect exists whenever any of the simulation threads is context switched out, leading to the type of slowdown that we observe.

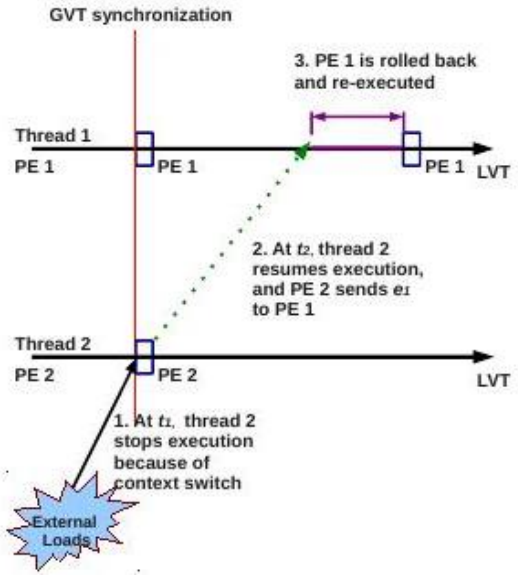
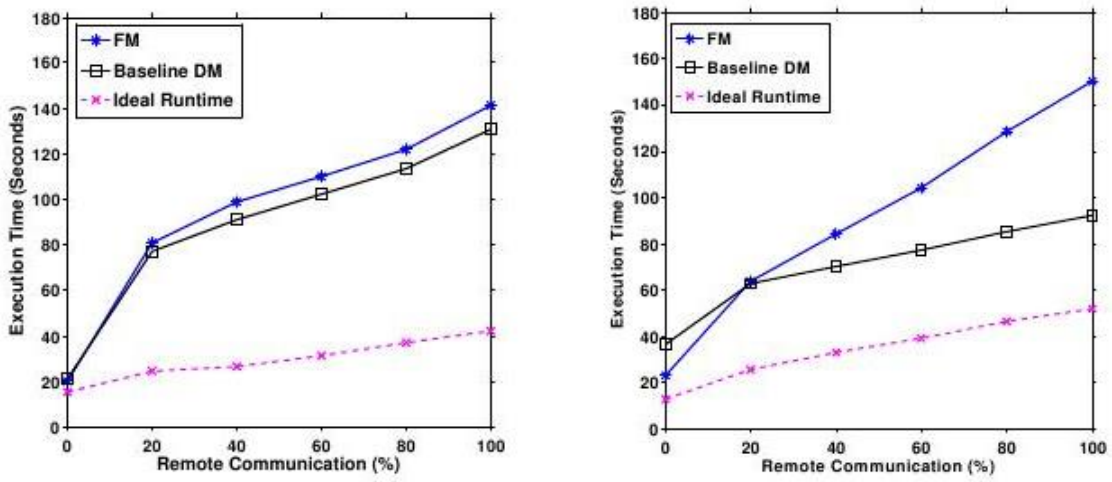


Figure 12 - A Rollback caused by Interferences from External Loads

Note that the effect also holds if we use conservative simulation. A thread that is context switched out is not able to update the lookahead at other LPs, preventing them from proceeding. In fact, this problem generalizes to any parallel application with dependencies.

3.5.3 Can Dynamic Mapping Help?



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 13 - Performance of FM vs. Baseline DM (Interfered by 1 External Load)

To address the destructive behavior that occurs in the presence of interference, we first attempt dynamic mapping (DM) of threads to PEs. More specifically, in this scheme, we periodically remap the threads to different PEs (recall that each PE encapsulates a group of objects in the simulation). The intuition behind DM is that it allows active threads to rotate across the different PEs, avoiding having a PE lag far behind the others.

Recall that each thread in ROSS-MT executes a loop that repeatedly performs the simulation tasks such as sending and receiving events and event processing. To implement DM, we add a new step at the beginning of the loop where a thread determines which PE to associate itself with; the base implementation simply rotates threads in a round-robin fashion across the PEs. Consider the example as shown in Figure 12. After thread 2 finishes the execution of PE 2 for an iteration, it then switches to PE 1. Thus, in principle, the active thread alternates working on PE 1 and PE 2, reducing the LVT difference between them. Alternative basis for scheduling PEs to threads are possible (for example, attempting to work on the PE with the lowest LVT).

Note that a side-effect of remapping threads to PEs is a loss of data locality: FM permanently maps a hardware thread to a unit of work, and the caches for the core are populated with the data relevant to it. As DM remaps work across cores, the PE data must be brought to each new core (from shared lower level caches or main memory).

A second, more serious, limitation of DM is its limited opportunity for assisting performance. More precisely, for correctness, two threads cannot be attached to the same PE

concurrently, which prevents remapping from being able to assist if the context switched thread happens to hold the lock on the PE. We implemented efficient synchronization using a condition variable and a spin lock for each PE. More precisely, a PE status is checked (without locking); if the status is busy, the thread moves on to the next PE. If the status is free, we acquire the spin lock for the PE, check if it is still free and set it to busy if it is. The thread is admitted to work on the PE. Once the iteration is over, it sets the PE status to free and moves on again to the next PE.

Thus, DM is limited if the first thread is switched out while in the middle of processing a batch since the PE will be marked as busy until the thread is scheduled again. Since this is the common case, DM cannot effectively solve the problem. Figure 13 shows the performance of original FM ROSS-MT in comparison to the baseline DM version for both the Intel Core i7 (Figure 13(a)) and the AMD Magny-Cours (Figure 13(b)) platforms. In particular, the entire simulation is interfered by 1 external load. We find that DM achieves up to 10% performance improvement over FM on the Intel Core i7 platform. Moreover, DM can achieve better performance than FM on the Magny-Cours only under high remote communication (> 20%). The gap remains substantial with respect to proportional slowdown (Equation 2).

Table 9 - Efficiency of FM vs. Baseline DM on the Intel Core i7 System (Interfered by 1 External Load)

Remote Comm (%)	20	40	60	80	100
FM	49.9%	47.1%	47.8%	48.6%	46.7%
Baseline DM	50.7%	49.5%	49.8%	51.2%	49.7%

Table 10 - Efficiency of FM vs. Baseline DM on the AMD Magny-Cours System (Interfered by 1 External Load)

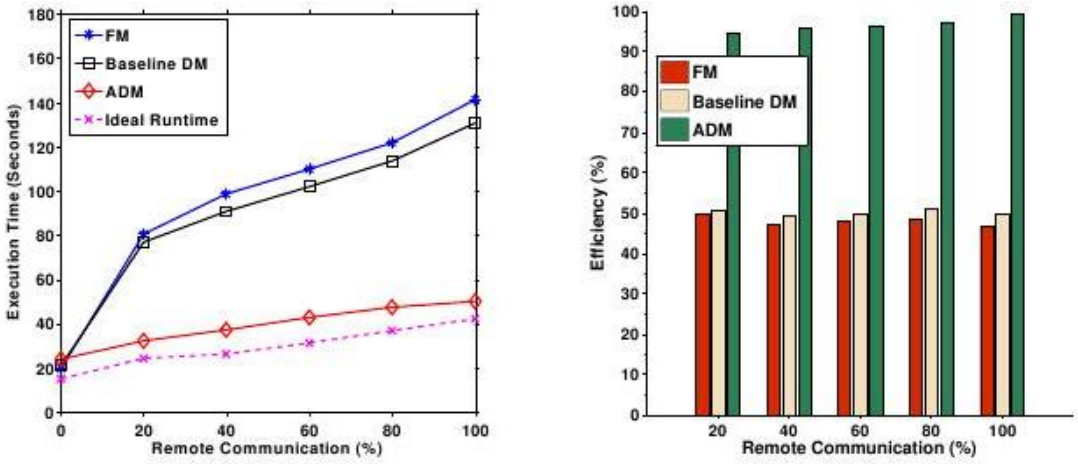
Remote Comm (%)	20	40	60	80	100
FM	67.0%	62.9%	61.1%	58.5%	56.8%
Baseline DM	87.3%	87.9%	88.5%	88.3%	88.3%

The efficiency of a simulation interfered by 1 external load is shown for both the Intel Core it (Table 9) and the AMD Magny-Cours (Table 10) platforms. While the results of AMD Magny-Cours system show improvements in efficiency of the baseline DM in comparison to the FM version are observed, the efficiency remains low especially for the Core i7. In most cases, DM was not able to help because the context switched thread held the PE lock.

3.5.4 Locality-Aware Adaptive DM

DM offers only limited relief from the slowdown experienced in the presence of interference. In addition, DM experiences poor cache locality, because of transient short term association between

threads and PEs. In this section, we propose a locality-aware adaptive DM (LADM) scheduler that is capable of addressing both limitations of DM.



(a) Execution Time (b) Efficiency

Figure 14 - Performance of ADM on the Intel Core i7 System (Interfered by 1 External Load)

3.5.5 Adapting the Number of Threads

The first improvement to DM, which we call adaptive DM (ADM), adjusts the number of work threads to the available hardware contexts: when a noise process is detected, the number of active threads is reduced to avoid experiencing expensive context switches. Thus, only active threads are allowed to execute PEs. Supporting ADM requires two main mechanisms: one to detect the presence of interference, and another to adjust the number of active threads. Finally, a third mechanism is required to check if the interference is no longer there and to reactivate idle threads. We discuss these mechanisms in the remainder of this subsection.

The presence of noise is detected as follows. During execution, each active thread periodically monitors its total event processing time (the period is set to $\frac{T_{gvt}}{4}$ simulation loop iterations in our implementation, where T_{gvt} is the GVT interval). The *Average Processing Time per Event* (APTE) of each active thread is calculated by dividing the total event processing time by the corresponding number of processed events. A performance anomaly is decided if the rate of maximum APTE to minimum APTE is beyond a user-defined threshold. After the mechanism decides a performance anomaly, the status of the thread with maximum APTE will be configured as "inactive". Each thread checks its status at the beginning of the

simulation loop, and inactive threads idle. We discovered that the threshold plays an important role in the performance of simulation. If the value of the threshold is too large, then the performance anomaly is not reliably detected when interference from external loads exist. On the other hand, too small a value can cause ADM to incorrectly inactivate a PDES thread in an interference-free environment. We use a threshold of 1.8 which we empirically found to work effectively on both platforms.

The final mechanism checks if the noise has disappeared and hardware contexts are again available. One inactive thread is re-activated periodically. If noise remains present, then the deactivation logic detects that and deactivates the thread. Thus, the reactivation period must be significantly larger than the detection period to avoid too frequent testing: (we use $10 \times T_{gvt}$, 40 times larger than the detection period).

ADM can reduce the effect of interferences from external loads, and thus significantly improves the performance of the simulation in the presence of external loads. Consider a 48-way simulation interfered by 1 external load on the 48-core AMD Magny-Cours machine, for example. Once a performance anomaly is successfully detected, the simulation is then executed by 47 active threads. The OS scheduler will later assign each thread to a different core, thus reducing interferences between PDES threads and the external load.

3.5.6 Improving the Data Locality

To improve the data locality, we modified the ADM scheduler to increase locality: we call this implementation locality-aware adaptive dynamic-mapping (LADM). Similar to FM, at the initialization of simulation, each thread is assigned to a primary PE, and maintains this assignment in the absence of interference to maximize locality. Once interference is detected and a thread (or more) is deactivated, the PE assigned to the inactive thread a PE is marked as an orphan until such a time where its thread is reactivated. The remaining active threads divide their time between their primary PEs and orphan PEs.

In particular, after each event processing iteration on its primary PE, each active thread checks PE's on the orphan list in a round-robin fashion; it selects an orphan that is currently behind its primary PE in the number of processing iterations (alternatively, LVT may be used). The status of the selected PE is then checked, and the spin lock for it is acquired if its status is free. Once the thread is admitted to work on the PE, it executes N_{batch} iterations before switching

back to its primary PE. We set N_{batch} to 10 in our simulations. The thread returns to its primary PE if all the orphan PEs have caught up with it. Unlike ADM, the PEs whose primary thread is active remain exclusively processed by that thread, and only orphan PEs experience a loss of locality.

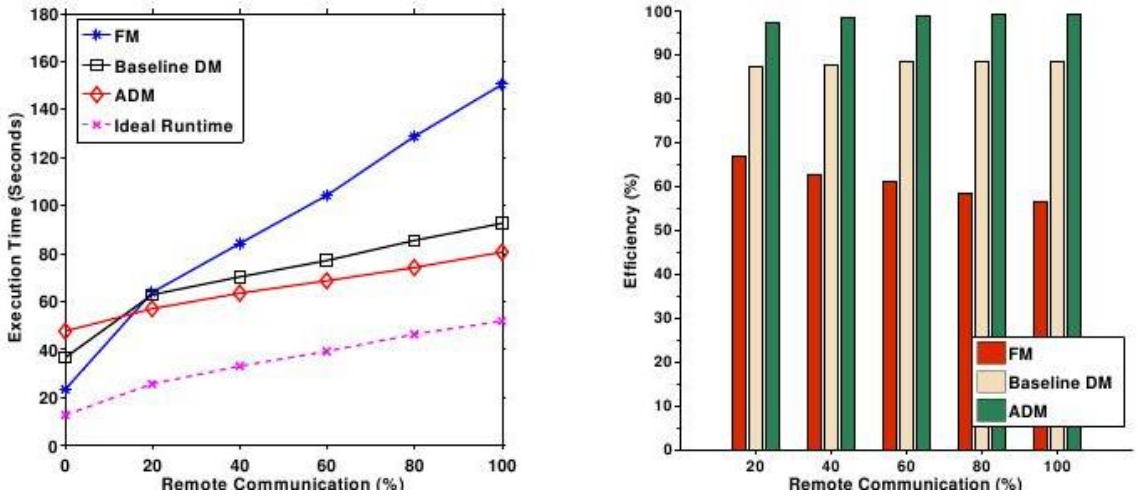
3.5.7 The Expected Runtime of LADM

Suppose that the simulator is configured with N_p threads at the initialization of simulation, where N_p equals with the total count of hardware threads on the multi-core platform. In addition, we divide the simulation into n small intervals $[X_{j-1}, X_j]$ indexed by j. Let N_j be the number of external loads running concurrently with PDES during the interval j of the simulation. Once LADM detects N_j ($N_j < N_p$) external loads, the simulation is then executed by ($N_p - N_j$) active threads during the interval j. The expected runtime of the entire simulation is thus approximated by:

$$T_{expected} = \sum_{j=1}^{n} \frac{N_p}{N_p - N_j} \times T_j \tag{3}$$

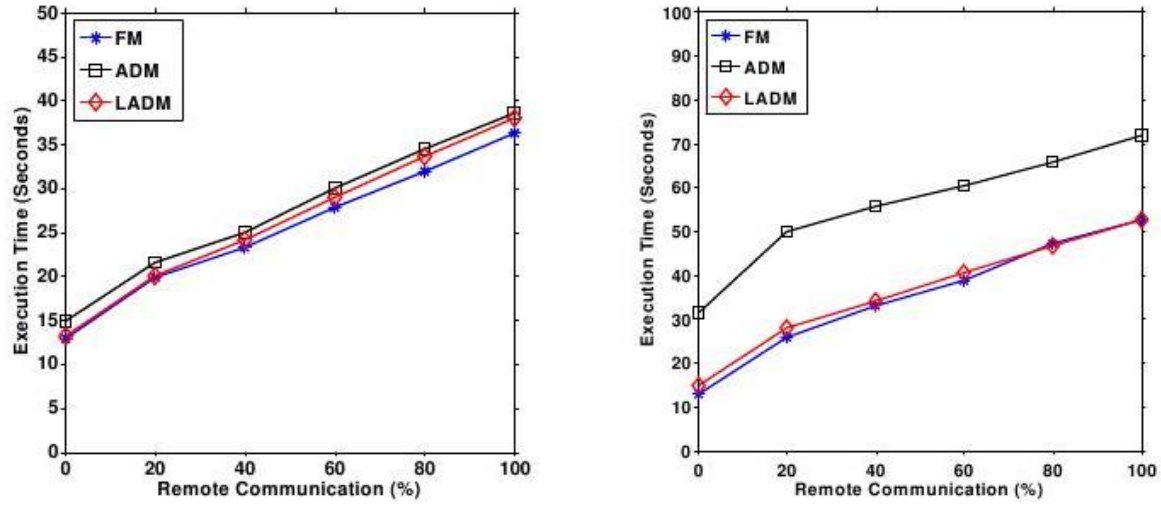
where T_j is the execution time required for the interval j of a FM simulation without interference. Moreover, LADM allows at least 1 active thread to execute the simulation if $N_j \geq N_p$.

It is important to note that LADM does not achieve proportional slowdown. ADM schedulers simply give up hardware contexts that are in contention to avoid a situation where they are context switched. Because of this conservative behavior, it is possible for interference loads to crowd- out the simulation threads resulting in significant slowdown under high interference. However, the OS scheduling policy will cause inefficient operation if more threads are running than there are available hardware contexts. To approach proportional slowdown, alternative OS scheduling policies are needed.



(a) Execution Time (b) Efficiency

Figure 15 - Performance of ADM on the AMD Magny-Cours System (Interfered by 1 External Load)



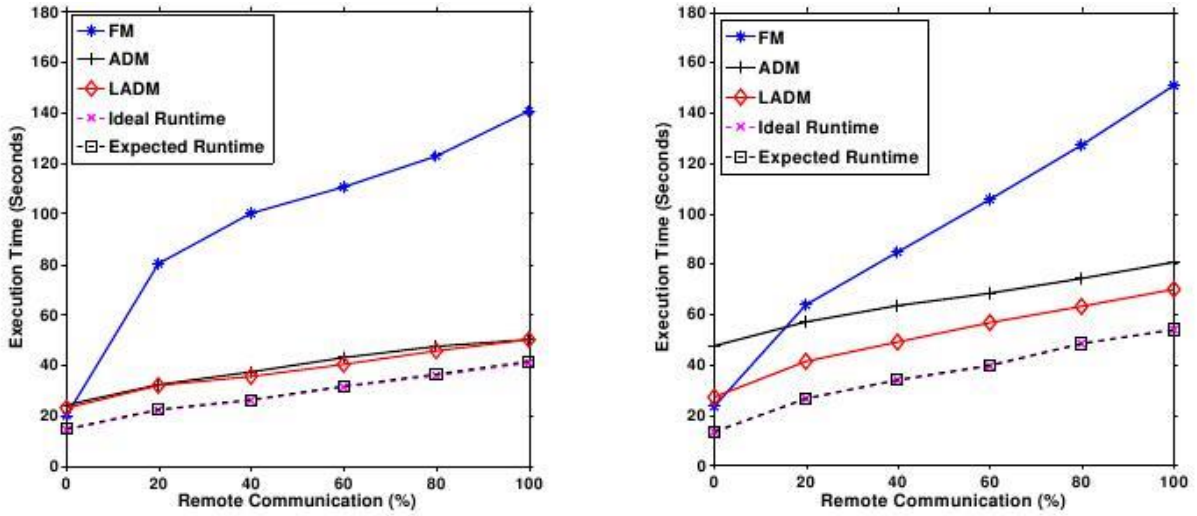
(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 16 - Performance of Locality-aware Adaptive Dynamic-Mapping Scheme (No External Load)

3.6 Related Work

In the general high performance computing community, it is well known that communication is a common performance bottleneck, especially for fine-grained parallel applications [60]. As a result, managing the impact of communication is a recurring focus of the parallel processing community. One of the approaches to reduce communication latency is to improve the network performance. For clusters, high performance networks [43] and networking abstractions,

communication libraries and system software implementations (e.g., [6, 18, 31]) have been proposed. In a multi- core environment, the design of the on-chip interconnect remains an open research problem [54]; most existing designs use the on-chip interconnect to implement indirect communication through shared caches.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 17 - Performance of Locality-aware Adaptive Dynamic-Mapping Scheme (1 External Load)

At the application/algorithm levels, there are general techniques for optimizing parallel applications such as overlapping computation and communication and reducing lock contention) [60, 1]. Efficient partitioning is necessary to reduce remote communication [66, 41]. However, most efforts in implementing parallel applications discover that application insights and awareness of the architecture are necessary to optimize the parallel implementation [58, 13, 20]. PDES is difficult to parallelize because of its fine-grained nature, and complex and dynamic dependency pattern [25], making it substantially different from typical parallel applications. Thus, we focus on optimizations specific to PDES, rather than other applications or parallel processing in general.

3.6.1 Optimizing Communication for PDES

Previous works have demonstrated the importance of partitioning to reduce the communication frequency in PDES (e.g., [41]). Similarly, dynamic partitioning and workload rebalancing mechanisms have been proposed to repartition the simulation to recover dynamic behavior changes of the simulation model for both conservative (e.g., [7]) and optimistic (e.g., [57]) synchronization protocols.

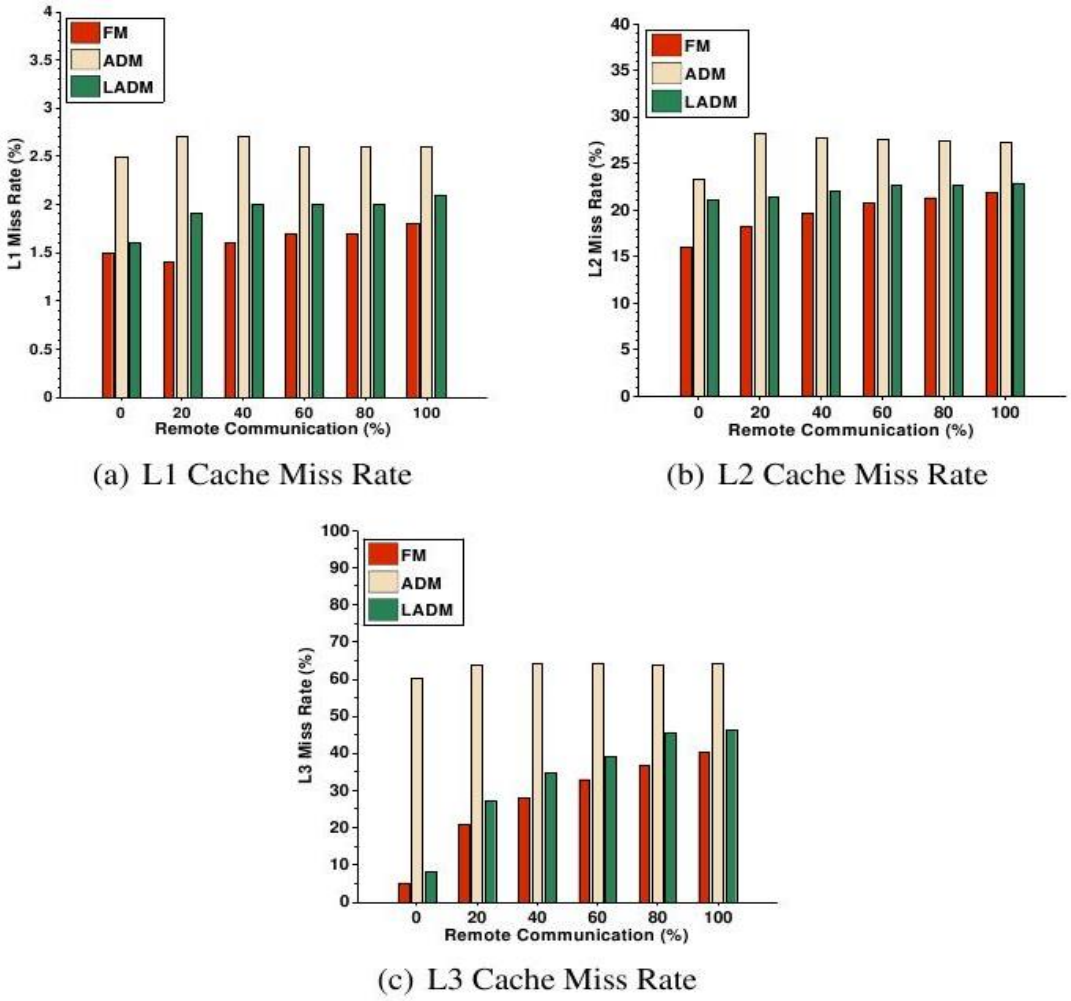


Figure 18 - Cache Performance of 48-way Simulation on the AMD Magny-Cours System

Chetlur et. al. [14] proposed the use of message aggregation, where multiple event messages are combined in a single communication message, to amortize the overheads associated with communication across multiple messages. Sharma et. al. [67] explored optimizing the polling frequency to check for the presence of event messages. Rajasekaran et. al. [61] explored using a single anti-message with the earliest rollback time-stamp to inform receiver PEs of rollbacks instead of sending individual anti-messages for each remote event to be cancelled. Mattern developed a non-blocking GVT algorithm which allows event processing to proceed concurrently with GVT computation, allowing the cost of that expensive operation, which includes global communication among the PEs, to be hidden [46]. Fujimoto et. al. designed the rollback chip, which implements wolf-calls (very fast notification of all PEs of the

occurrence of a rollback) [29]. Wolf-calls substantially limit cascading rollbacks that occur due to communication latency. Noronha et. al. used a programmable network card to optimize event communication and GVT computation [53].

3.6.2 Shared Memory PDES

Fujimoto's GTW simulator is one of the first shared memory optimistic PDES implementations. It exploits shared memory for efficient message communication. GTW also implemented optimizations such as direct cancellation, which allows an LP to cancel out erroneously sent remote events directly, eliminating the need for anti-messages [19]. Similarly, in shared memory, messages can simply be written into a buffer and become visible to all processors. The exploitation of such features allows the GVT computation algorithm to be implemented through a single round of inter-processor communication (as opposed to at least two rounds required for message passing programming model) with minimal number of shared variables and data structures. Fujimoto and Hybinette also describe an efficient on-the-fly fossil collection algorithm to enable fast reclamation of memory [28]. They also explore efficient buffer management algorithms for shared memory environments [27].

While some of the schemes developed for shared memory, which were developed and evaluated on Symmetric Multi-processor machines, can apply for manycores, the tighter coupling of processing elements and shorter communication delays for accessing shared caches requires at least a careful reconsideration of these approaches. Our paper reports on experiences in optimizing a PDES simulator on emerging multicore systems.

Our work is targeted towards emerging multi-core and many-core architectures. Current examples of these architectures commonly employ chips with multiple-cores with on chip memory controllers such as the AMD Opteron Magny-Cours. Reference [47] provides insights into NUMA related performance issues on such multi-core platforms and also discusses commonly employed solutions to these problems. Our multi-threaded implementation can be much more beneficial on these platforms provided the design is NUMA-aware. Some of the techniques presented in [47] were incorporated in our work such as use of first-touch policy for memory allocations.

3.6.3 Prior Work on Partitioning

Several researchers have studied the use of partitioning to optimize the performance of PDES. In

this section, we review some of the most related works.

Several efforts have targeted partitioning for logic simulation based on the static model topology. For example, Cloutier [16] studies the impact of various partitioning techniques on the performance of time-warp simulation of logic circuits. Various circuit parameters are used in partitioning including circuit topology (the netlist), the gate delays, the relative number of evaluations of the model of each circuit element, and the relative complexity of the element models evaluation. For partitioning, they represent a logic circuit by a weighted directed acyclic hypergraph. In the first approach, the computation load is distributed to the processors. The second approach is a mincut algorithm for weighted hypergraphs which minimizes the communication load of the simulation.

Another aspect of Static Partitioning is Static Analysis, obtaining the structural information from the model. This structural information can be more easily extracted from models written with certain design methodology or higher level design languages rather than the one written in general purpose programming languages. Reference [37] extracts such hierarchical structure information from DEVS models while [42] takes advantage of the design hierarchy of VLSI modules in Verilog. Instead of partitioning a gate-level netlist, [42] proposes a design-driven iterative partitioning algorithm which takes advantage of the design-level hierarchy embedded in VLSI module instances. A Verilog instance is represented by a vertex in the circuit hypergraph and is flattened to the gate-level if the load balancing was not achieved by instance level partitioning.

Similar to our work, Nandy et. al. [51] attempt to track model activity but use it for dynamic object migration. They represent the LPs as nodes with weight indicating expected execution time for activations of that LP and links represent communication channels with weight indicating the expected number of messages. Estimates are obtained by pre-simulation runs for the sake of abstracting the problem. They motivate the need for partitioning by presenting a benefit of 20% and the impact of load imbalance. Though the remaining paper focuses on a parallel partitioning scheme based on movement of nodes which yields as good a partition as a sequential scheme, the overall graph based abstraction of the simulation is a recurring idea in many later load balancing studies. Reference [8] presents a static partitioning and mapping algorithm for conservative parallel simulations. It assumes the same abstraction mentioned in [51], and presents a partitioning scheme based on Simulated Annealing.

Wilson et al. [76] is a good example of load balancing study in the context of optimistic PDES simulation. Emphasizing automation of partitioning decision they compare three possibilities of balancing workload among the processors. The first approach determines the object placement based only on the computation weight of each object. This approach concentrates on balancing the workload on the processors but ignores communication costs. The second approach, arrange the LPs in a linear chain so as to keep heavily communicating LPs close to each other in the chain, thereby increasing the chance that they will be assigned to the same subchain and hoping that computation and communication are not correlated. The third approach modifies the linear chaining algorithm to discourage clustering of extremes: pairs of computationally heavy-weight objects or pairs of computationally light-weight objects.

A more comprehensive approach to partitioning for distributed simulations is proposed in [23]. It presents a partitioning layer in JAMES II, a modeling and simulation framework. The partitioning process involves a model analyzer to extract the model properties and an infrastructure analyzer to obtain the underlying topology of computing platform. Partitioning algorithms then make use of both sets of information for mapping the model on the processors.

Thulasidasan et. al. [70] argue that dynamic load balancing presents significant implementation challenges due to object migration and explores the possibility of static partitioning for conservative simulation specifically for spatially clustered models with geographic hot-spots where most of the computation and messaging occurs (e.g. urban regions in transportation networks). They argue that in such models CPU load is a greater determinant of parallel simulation than message passing overhead. In general, dynamic object migration (e.g., [57, 7]) moves objects during run-time to achieve better partitioning. Since object migration is done on-line, with limited local knowledge, it cannot achieve the effectiveness of dynamic partitioning. However, unlike partitioning, object migration can adapt to changing simulation behavior.

3.6.4 Prior Work on Resilience to Interference

In this section, we first overview some prior works in the context of PDES. We follow this by describing the interference problem in the general parallel processing community.

Dynamic load-balancing Approaches for PDES

Dynamic load-balancing approaches rely on a monitoring scheme to detect load imbalance, and make dynamic adjustment to improve the performance of simulation. These approaches differ in metrics of detecting load imbalance, and balancing schemes.

Vitali et. al. [74, 73] present a load-sharing scheme developed for a symmetric multi-threaded optimistic PDES simulator. Each PE is executed by multiple worker threads, in order to improve parallelism of the simulation. The approach works by allowing a PE that is lagging behind to acquire additional threads to assist with its computation. Thus, the approach is on the face of it similar to our approach in that threads can be redirected to work on lagging PEs. The approach can effectively foster load balanced simulation, but cannot effectively solve the interference problem, as other threads cannot assist when threads keep getting context switched in the middle of event processing.

Wilsey et. al. [15] proposed a different approach to support run-time core frequency adjustment on many-core systems, with the goal of accelerating the critical path of execution of the Time Warp simulation. To balance workloads of LPs, the cores containing LPs with larger rollbacks are underclocked, while the cores having LPs with smaller rollbacks are overclocked. Though this approach may reduce rollbacks caused by external loads, the performance issue caused by the interference still exists as LPs can't advance if their executing thread is switched out.

Carothers et. al. [10, 12] designed a scheme to support background execution of Time Warp. A background central process periodically monitors the workload of each processor, and dynamically determines the set of processors to be used for the Time Warp Simulation. LPs are then distributed across these processors, by using object migration which is widely used in many existing dynamic load-balancing approaches [65, 21, 44]. Dynamic object migration cannot solve the interference problem as well unless all objects are migrated away from a context switched thread.

Other Approaches to Reduce the Effect of Interference on PDES

In this project, we demonstrate that the optimistic fixed-mapping PDES simulation kernel can suffer considerably in the presence of interference on the multi-core platforms, due to excessive rollbacks being generated. Malik et. al. [45] observed the same behavior present in the

cloud environment. To reduce excessive rollbacks caused by interference, they developed a protocol, called *TW-SMIP*, with the goal of identifying straggler messages early and thus avoiding frequent rollbacks.

Replication is another approach that is capable of reducing the effect of interference. As presented in [68], multiple copies of PDES simulation are executed simultaneously on heterogeneous workstation cluster. It allows the runtime reconfiguration in terms of runtime resource availability, and thus this approach can adapt to interferences from external loads.

Interference in General Parallel Processing

Similar to PDES, most parallel applications have dependencies between executing threads. Thus, when the interference occurs, active threads have to wait for context switched ones before continuing to execute and the pace of the execution is determined by the slowest thread. As a result, the performance of these applications can be substantially harmed [52, 72, 59]. Two approaches are widely used to balance workloads of threads at run-time: *work-sharing* and *work-stealing*. In work-sharing, when a thread completes its task, it grabs a new one from a central work pool shared across all threads [1]. In contrast, in work-stealing scheme, once a thread finishes its tasks, it steals other threads' tasks [24]. However, neither approach can solve the interference problem unless a context switched thread does not hold any task.

4.0 Evaluation Methodology, Results and Discussions

4.1 Performance Evaluation of ROSS-MT

In this section, we present a performance evaluation of ROSS-MT, including the three proposed optimizations. First, we discuss the evaluation environment, and the simulation benchmark that we use.

4.1.1 Experimental Setup and Benchmark

To capture a wide range of application characteristics we developed a synthetic, controllable benchmark that is a variant of the classical Phold benchmark. Phold is the most widely used for performance evaluation of PDES systems. The model starts with a number of objects that have events. Event execution sends a message to another object (picked uniformly among all the objects in the simulation). The message causes this object in turn to later send another event message to a third object. Thus, the number of events in the simulation remains constant. While Phold has a number of drawbacks: it's perfectly load balanced, with a flat dependency pattern, it is valuable for the characterizing the performance of the communication behavior of the simulator.

In particular, we modified Phold to allow control of the target probability for the events to allow us to control the percentage of events that are generated local to a core, to another core on the same machine, or remotely to a core on a different machine. This benchmark is similar to that used by Perumalla [56] and Bauer et. al. [5] in recent scalability studies of PDES on the IBM Blue Gene.

We evaluated performance of multi-threaded ROSS against MPI based ROSS on two hardware platforms:

1. A 4-core (8 thread) Intel Core i7-860 processor with 8 GB memory and Debian 6.0.2 with Linux version 3.0.0-1. Each core has a private 32KB L1 data and instruction cache and private L2 256KB cache. 8 MB L3 cache is shared among all the cores. We use classic PHOLD as benchmark application with configuration of 1000 LPs per PE, one PE per MPI node and total 8 MPI nodes. Simulation model thus consists of 8000 LPs distributed equally on 8 PEs and a PE is pinned to one hyperthread. We selected efficient settings for GVT computation period to balance fossil collection overhead

with rollbacks as per the experiment [5]. We use 16 KPs per PE and a GVT interval 256 for Intel Core i7 and 512 for AMD Opteron with batch size 8.

2. In order to verify the scalability of multithreaded ROSS on upcoming manycore platforms we studied multithreaded ROSS behavior on an AMD Opteron 6100 (Magny-Cours) 48 core machine (4 chips with 12 cores each) [17]. The chips are connected using Hyper-transport 3.0 links. Each chip consists of 2 dies and each die has 6 cores, with a 6 MB L3 cache shared among the cores on each die. Each core has private 64KB L1 and 512 KB L2 caches. The Hyper-transport links are cache coherent, thus 4P configuration provides 48 core shared memory environment. The server is running Ubuntu 10.10 with Linux version 2.6.35-30-server and has 64GB memory.

4.1.2 ROSS-MT Performance Analysis

We use the Clustered Phold model with 1000 objects per core. We fix the GVT interval at 512 (as recommended by prior evaluation studies of ROSS [5] and confirmed by our own experiments). Execution time is measured at different remote percentage for fixed batch size. We observed that batch size of 8 is optimal for both multi-threaded ROSS and MPI-based ROSS.

We implemented the three optimizations discussed in the previous section (barrier optimization, NUMA aware memory pool management, and distributed input queue). Figure 19 shows the performance improvement obtained from each of the optimizations in isolation and combined on the Core i7. We consider a 2-way, 4-way and 8-way simulation, while keeping the number of objects per thread the same.

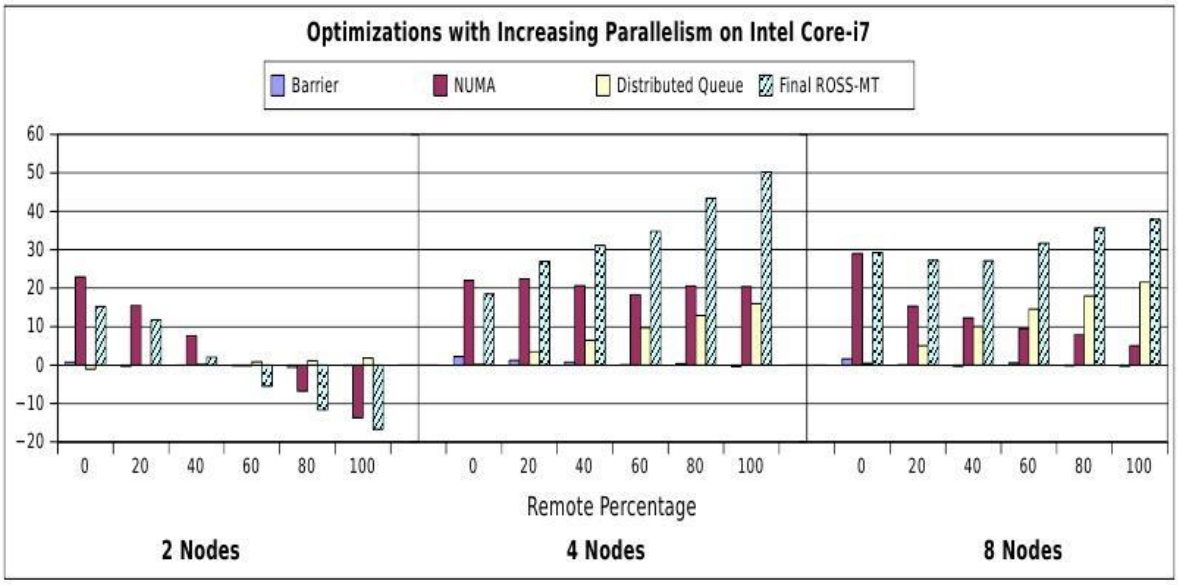


Figure 19 - Optimizing ROSS-MT on Intel core-i7

A number of observations stand out. For two nodes, as the number of remote messages increase, the optimizations are actually counterproductive. Queue distribution is not beneficial since the contention degree is not reduced, but the overhead is increased. Moreover, NUMA issues are not important either since each memory element is local to either of the two threads. Finally, lock contention issues are minor in the barrier implementation. It is interesting to see some gain initially, but that is likely due to the LIFO strategy introduced as part of the NUMA optimization; other optimizations are likely to introduce overhead without benefit for a two thread simulation. As the number of threads is increased, the optimizations start to become useful. The baseline barrier implementation seems efficient up to 8 nodes; the optimized implementation does not result in significant improvement in performance. The optimizations seem to interact positively as their combined effect is higher than the linear sum of their isolated effects; up to 50% improvement relative to the baseline ROSS-MT is achieved.

Figure 20 shows the breakdown of the execution time among the various stages of the simulation. We note that with ROSS-MT, significantly smaller percentage of time is spent in event send and receive (30% compared to 50% for MPI). Less time is also spent in GVT computation (18% compared to 25%). Moreover, the percentage of time spent in event processing is more than doubled.

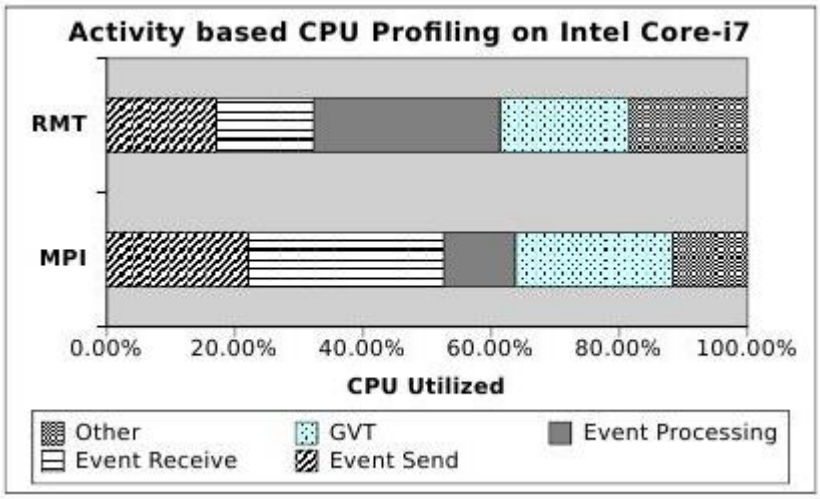


Figure 20 - Execution Time Breakdown – Core i7

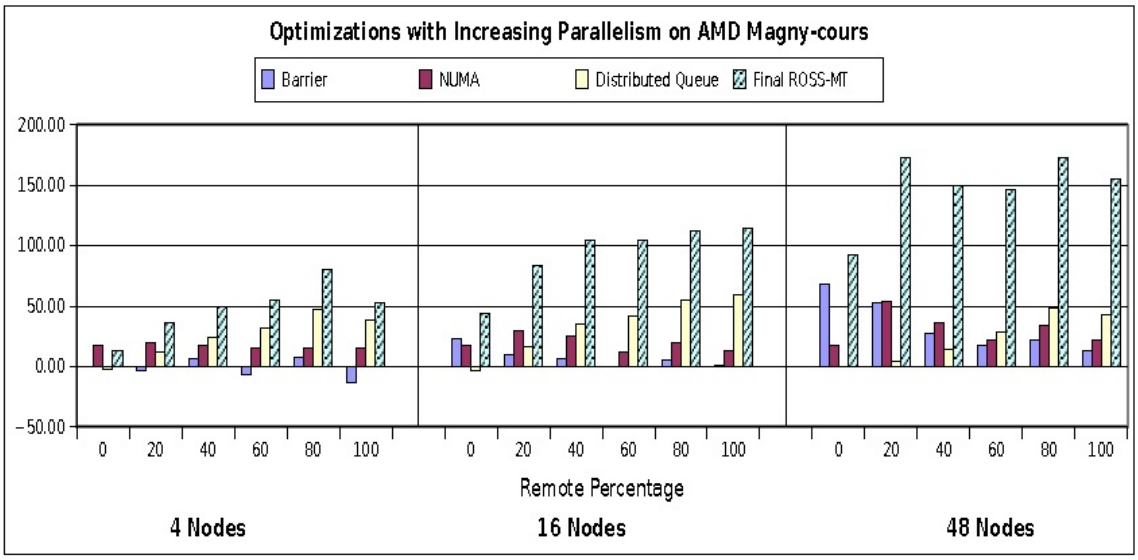


Figure 21 - Magny-Cours performance for different degrees of parallelism

Figure 21 shows the impact of the optimizations for the Magny-Cours machine for 4, 16 and 48 thread scenarios. Since the bottlenecks were most severe for this machine, the optimizations yield substantial improvement in performance (over 150% for 48 threads). The impact of the barrier optimization increases with the degree of parallelism, and reduces slightly with the increase in event communication (recall that the barrier optimization affects GVT computation but not event communication). Like the Core i7, the queue distribution benefit increases as contention on the queue increases: both with increasing the degree of parallelism and increasing the remote event percentages.

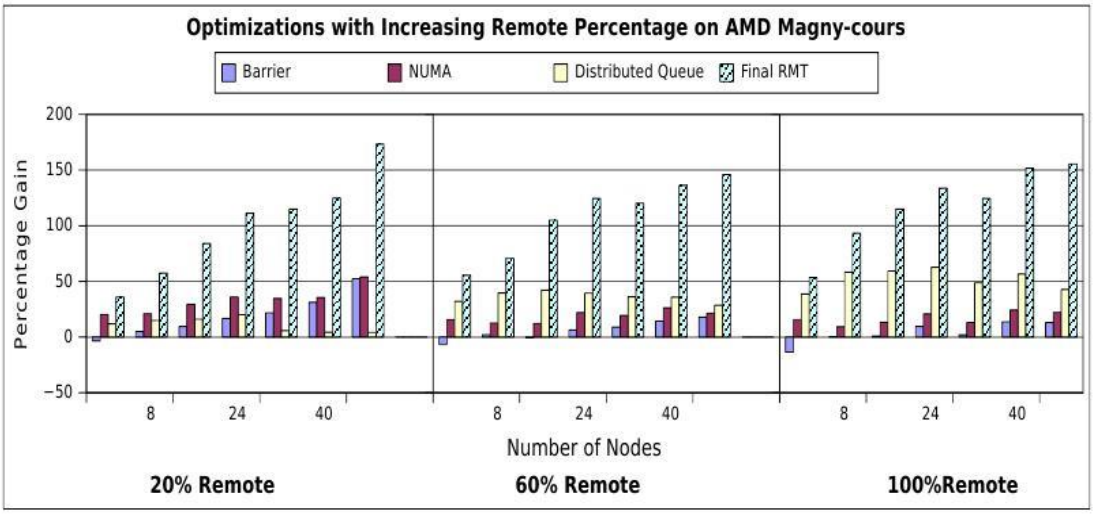


Figure 22 - Magny-Cours Performance as a function of Remote Events

In Figure 22 we show another snapshot of the Magny-Cours machine performance. In this case, we fix the percentage of remote communication and show the performance improvement for varying degrees of parallelism. In general, the performance benefit of the individual optimizations increases with the degree of parallelism.

The results show that the barrier and NUMA optimizations significantly improve performance when remote communication is infrequent. In such models, the LIFO strategy in the NUMA optimization likely leads to much better cache locality explaining the better performance. As can be seen in Figure 20, GVT computation consumes a substantial portion of execution time in the 20% remote case; the barrier optimization impacts GVT computation and significantly improves performance. Finally, the distributed queue optimization increases in impact as the percentage of remote communication increases, increasing the pressure on the input queue. Note that in these figures, we show the speedup (rather than reduction in run-time); this explains why their combined effect is high since it is bound by the product rather than the sum of the speedup from the individual optimizations.

Figure 23 shows the performance improvement for the Magny-Cours ROSS-MT with all optimizations relative to the baseline ROSS with MPI. The performance improvement increases as the percentage of remote events increases (increasing the use of the communication subsystem and the optimizations). However, in general, the benefit is less pronounced as the degree of parallelism is increased. We believe that with some effort, we can track down additional lock contention issues and address them.

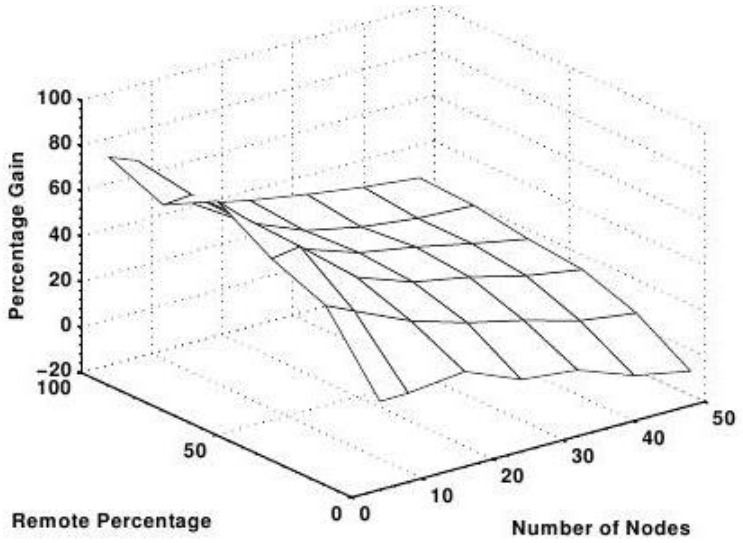


Figure 23 - Performance Improvement Relative to MPI – Magny-Cours

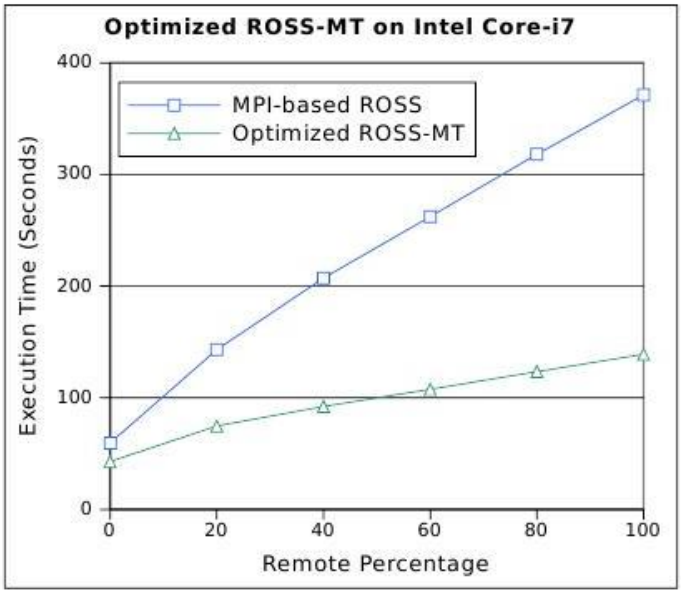


Figure 24 - Execution time of the optimized ROSS-MT on Intel core-i7

We show the impact of the optimizations on run time on the Core i7 machine with 8 cores and event message size of 8 bytes in Figure 24. It's clear that the multi-threaded implementation is substantially faster than the MPI version on this platform. Figure 25 shows the same comparison for the Magny-Cours platform with 48 cores (same message size). ROSS-MT also outperforms the MPI version, although the gap is substantially closer.

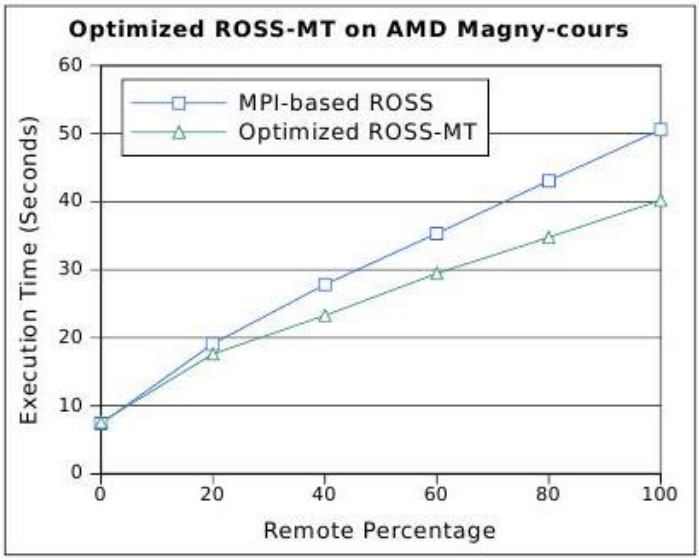


Figure 25 - Execution time of the optimized ROSS-MT on AMD Magny-Cours

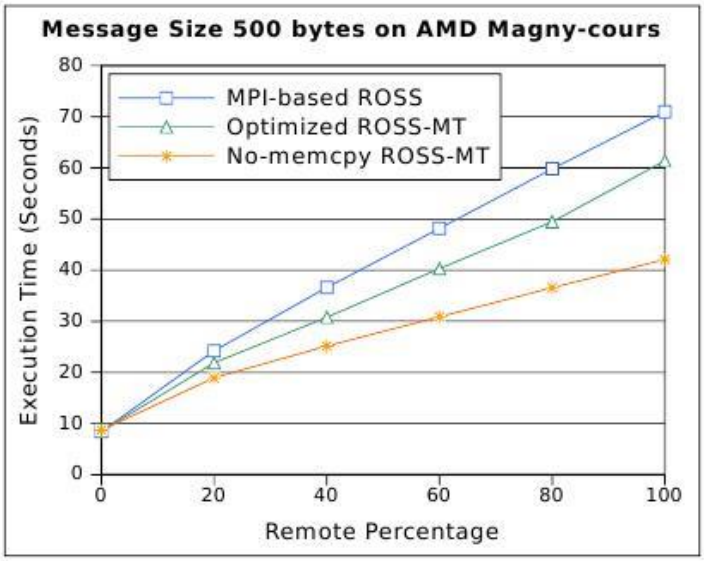


Figure 26 – Execution time of the optimized ROSS-MT on AMD Magny-Cours with message size 500 bytes

In the next experiment, we study the impact of bigger message sizes on the performance of the simulator. The message size is a key factor impacting communication cost; not only is transmission cost increased, but buffer copies and checksum operations increase in cost with the size of the message. We studied the impact of message size on event rate of MPI based ROSS and ROSS- MT. Figure 26 shows the performance of ROSS-MT on the AMD Magny-Cours with message size 500. The multi-threaded implementation does a message copy into the input queue to keep a separate copy in case of rollback. However, this additional copy can be avoided by keeping pointers and tracking sharing to the message. We implemented a preliminary version of

this optimization (results shown in Figure 12); substantial reduction in execution time is observed, making the ROSS-MT as much as 65% faster than the MPI implementation. We believe that there remains room for ROSS-MT optimization.

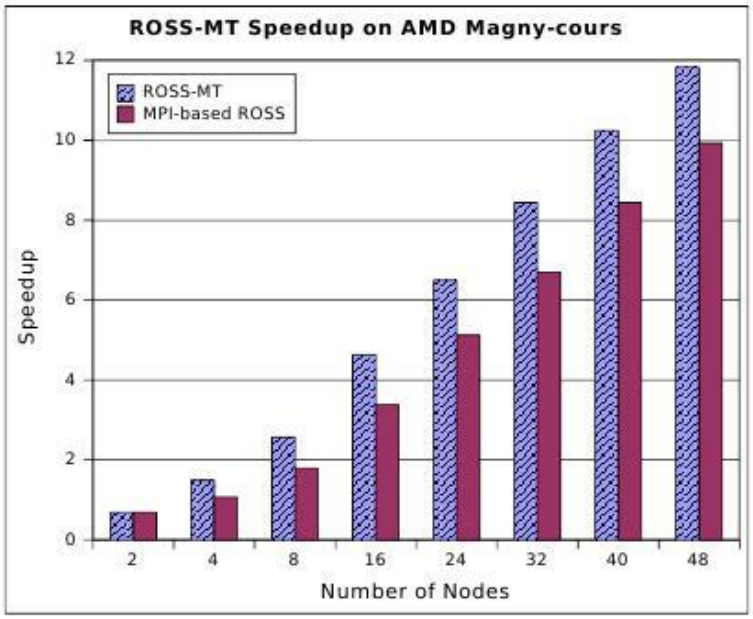


Figure 27 - Speedup for AMD Magny-Cours

Finally, Figure 27 shows the speedup that ROSS-MT and MPI ROSS relative to sequential execution as the number of cores is increased with 100% remote communication. Speedup increases linearly up to the full 48-cores for both implementations where ROSS-MT achieves speedup close to 12, while the MPI version achieves a speedup of 10. In summary, we showed the performance of ROSS-MT on two different multi-core platforms. The large difference in behavior can be explained by the following differences in their architecture.

1. Even though the number of cores is high on the Magny-Cours each core is individually less than the Intel Core i7 cores.

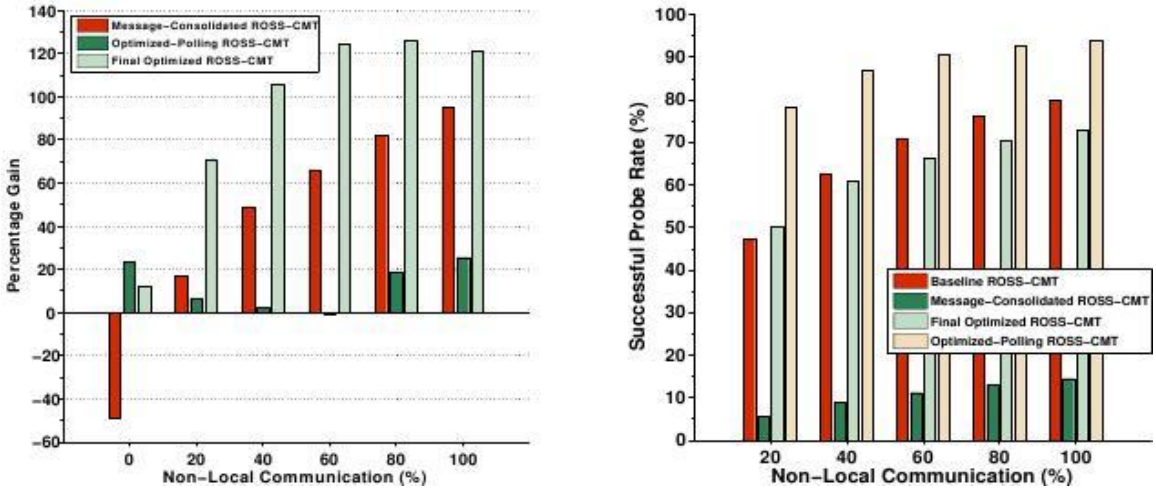
2. Although NUMA remote cache access issues have been improved by the NUMA optimization and LIFO free memory strategy, NUMA issues are still present and significant remote cache accesses occur in the current multithreaded ROSS implementation.

3. Further, only L3 is shared among the 6 cores on a die. On Core i7, all the cores share L3,

and two hyperthreads on the same core share L1 and L2.

4.2 Evaluating ROSS-MT on Multicore Clusters

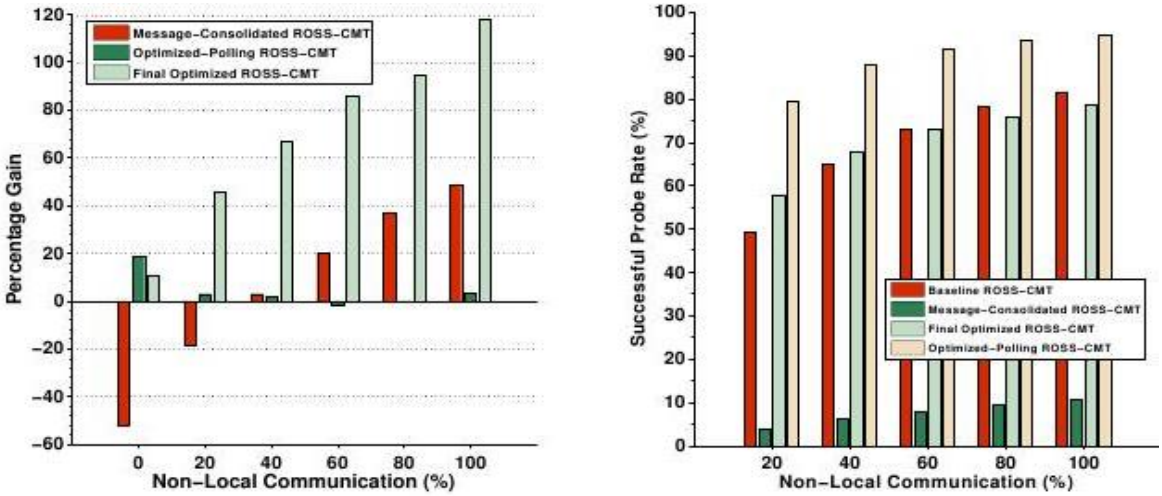
In this section, we study the effectiveness of the three proposed optimizations for ROSS-MT executed on CMs on the performance of PDES. Our results demonstrate that these optimizations allow us to mitigate the negative impact of slow communication links on PDES performance and scalability.



(a) Performance Improvement (against Baseline ROSS-CMT)

(b) Polling Success

Figure 28 - Infrequent Polling and Message Consolidation for 32-way PDES



(a) Performance Improvement (against Baseline ROSS-CMT)

(b) Polling Success

Figure 29 - Infrequent Polling and Message Consolidation for 64-way PDES

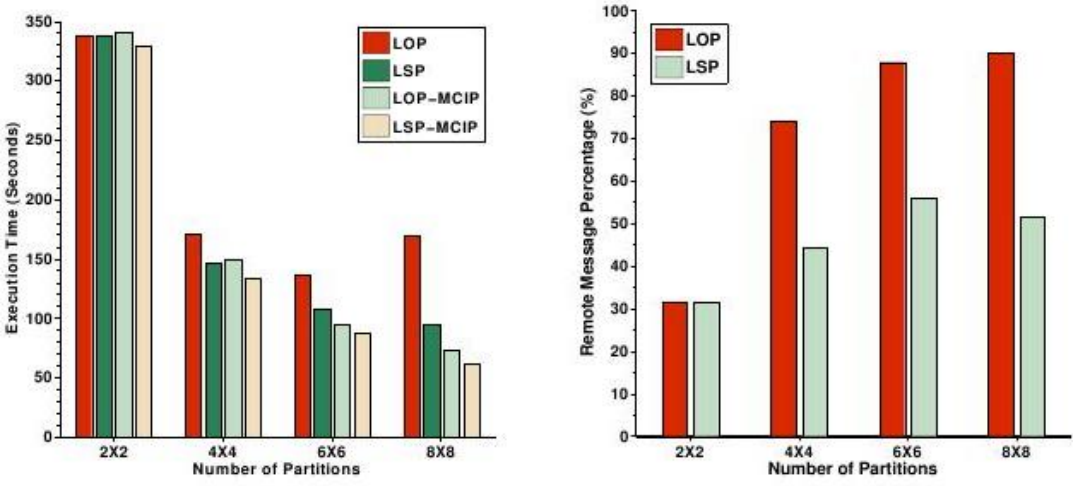
4.2.1 Message Consolidation

In our first experiment, we evaluate the performance of ROSS-CMT and ROSS-MPI with and without message consolidation, by using the classic Phold [26] benchmark, with 1000 objects for each PE. Figure 7 shows the impact of message consolidation on a 32-way optimistically synchronized simulation. In Figure 7(a), we see the run time as a function of the percentage of non-local messages (i.e., the percentage of Phold event targets that reside on another core; this includes cores on the same node, as well as cores on other nodes). As the non-local message percentage increases, message consolidation allows both versions of the simulation to improve their performance compared to the case without message consolidation. ROSS-CMT performs better than the MPI-based implementation with or without message consolidation. The benefits that ROSS-CMT obtains from message consolidation are significantly higher than those obtained by ROSS-MPI, because the CMT version can consolidate messages originating from any thread within the same node. In contrast, ROSS-MPI can only consolidate the messages from one process to another process, and as a result, it benefits from consolidation only slightly in the best case, and it is even harmed in some cases. These effects are shown in Figure 7(b), which depicts the average number of events that are consolidated into a message. Clearly, ROSS-CMT is able to consolidate a significantly higher number of messages eliminating the high per-message sending overhead. At 100% non-local communication, ROSS-CMT with the message consolidation is capable of providing a performance gain of 2X against the baseline ROSS-CMT, and 3X against ROSS-MPI.

In the next experiment, we show the impact of message consolidation on PDES scalability at 20% remote communication (Figure 8(a)), and 80% remote communication (Figure 8(b)) respectively. We fix the total number of objects (to 60480), and equally distribute them across the PEs. The simulation is performed on 2 nodes, 4 nodes, and 8 nodes respectively, with 8 hardware threads used for each node. Message consolidation achieves a performance gain of up to 2.2X against the baseline, and up to 2.5X against ROSS-MPI at 80% communication. We also discover that the message-consolidated ROSS-CMT performs worse than the baseline ROSS-CMT with the case of 8 nodes at 20% remote communication. This is because of a large number of unsuccessful probe operations, leading to our next optimization.

4.2.2 Infrequent Polling

In asynchronous parallel applications, a receiver does not know when to expect communication from a sender, and has to resort to polling. Polling is an expensive operation on communication channels that are inter-node. In our next experiment, we study the performance impact of infrequent polling strategy applied to PDES, since it is an asynchronous application. We investigated several polling periods, and selected 4 (poll after processing every 4th event) because it works well across a range of communication frequencies and simulation scales. We also investigated adapting the polling frequency, but the effect was minor. Polling also interplays with message consolidation, since consolidation reduces communication messages, increasing the chance of unsuccessful polls.



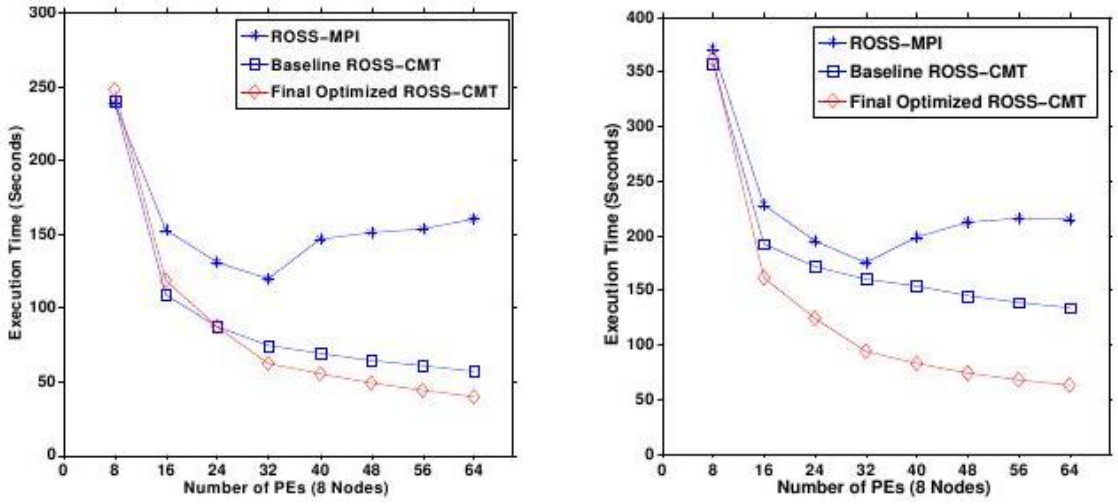
(a) Execution Time Comparison (b) Remote Message Percentage

Figure 30 - Performance Evaluation of Different Partitioning Strategies

Figure 28(a) shows the percentage gain in performance with infrequent polling, on its own, as well as when it is combined with message consolidation for a 32-way simulation. Somewhat surprisingly, there is a large performance drop of about 50% in message-consolidated ROSS-CMT at the non-local percentage of 0. We expected some performance loss since the message consolidation overhead is incurred, without finding opportunities for consolidation. While infrequent polling on its own results in modest improvements in performance (up to 20%), when combined with message consolidation it significantly improves performance (up to 120% relative to baseline). As message consolidation combines messages, they arrive less frequently, allowing infrequent polling to successfully eliminate message probes. Figure 28(b) shows the

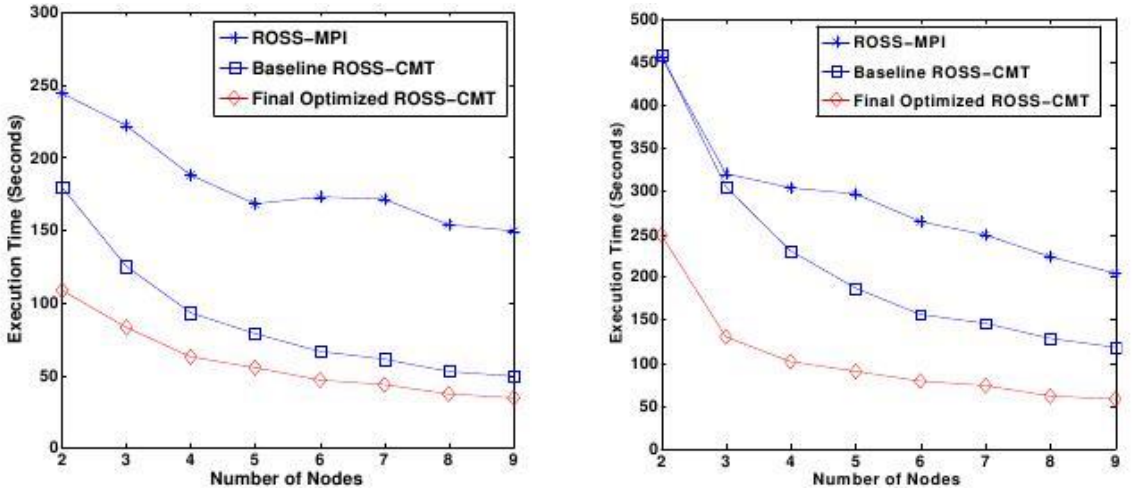
successful probe rate for each of the three simulator versions, as well as the baseline. We note that the baseline ROSS-CMT has a very good successful probe rate because of its frequent communication across the network as each communication point represents all the threads on that node. In contrast, ROSS-CMT with message consolidation only has much lower successful probe rate than the others, since message frequency is lower, but still has substantially shorter execution time (up to 90% improvement) than the baseline ROSS-CMT as shown in Figure 28(a). We repeated the experiment for a 64-way simulation (Figure 29(a) and Figure 29(b)), with similar results.

4.2.3 Latency-Sensitive Model Partitioning



(a) 20% Remote Communication (b) 80% Remote Communication

Figure 31 - PDES Scalability as Number of Hardware Threads per Node is Increased



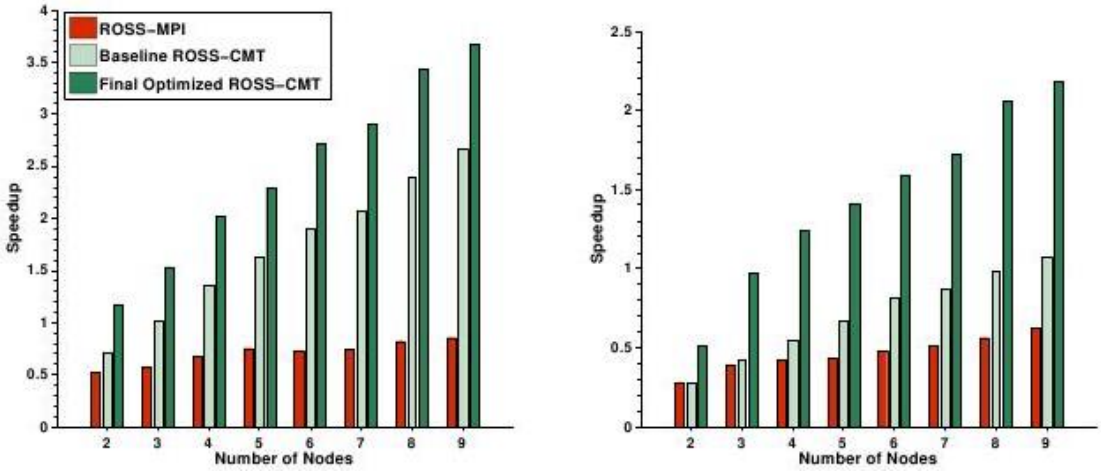
(a) 20% Remote Communication (b) 80% Remote Communication

Figure 32 - PDES Scalability for Different Number of Nodes

Figure 30 shows the impact of partitioning sensitive to heterogeneous latencies on CMs. In Figure 30, the number of partitions *N X T* indicates that *N* nodes are used with T threads per node. Each partition is mapped to one thread. This experiment uses a hierarchical Phold model [2]. The model is initialized with 4 initial events per object. In our experiment, four types of partitioning strategies are compared. A *Latency-Sensitive Partitioning* (LSP) computes a node-level partition first and then each part is partitioned again at the core level. This strategy ensures that the groups of objects with most communication are placed on the same node. It is compared with a *Latency-Oblivious Partitioning* (LOP) where the latency heterogeneity is ignored and the partitioning tool tries to minimize all communication between cores. The remaining two strategies combine the partitioning with both Message Consolidation and Infrequent Polling (MCIP).

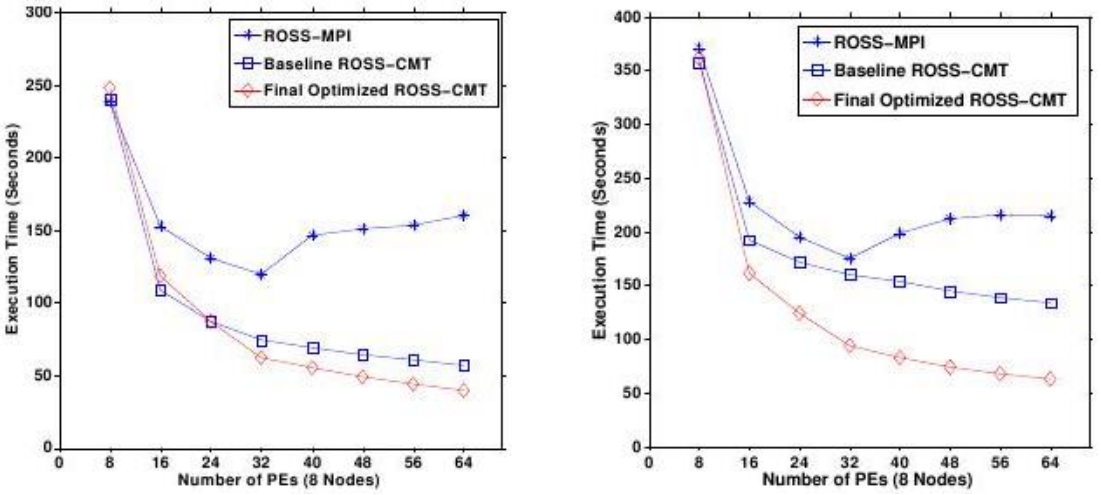
Figure 30(a) demonstrates the impact of each of these strategies with the hierarchical model described above. It shows that the benefit of LSP increases as the number of partitions increases. LSP performs 15% better than LOP in the case of 16 partitions while up to 44% better in the case of 64 partitions. In addition, MCIP optimization is orthogonal to partitioning strategies and significantly improves the performance both in the case of LSP and LOP. Figure 30(a) also indicates that LSP and MCIP optimizations when used together (LSP-MCIP) provide the best performance of these 4 strategies compared here. Figure 30(b) shows the percentage of remote messages across nodes out of the total messages among partitions in the case of LSP and LOP. LSP places the most communicating object groups on the same node as indicated by the

significant reduction in the percentage of remote messages.



(a) 20% Remote Communication (b) 80% Remote Communication

Figure 33 - Speedup of PDES Optimistic Simulation against Sequential Simulation



(a) 20% Remote Communication (b) 80% Remote Communication

Figure 34 - Impact of Event Process Granularity (EPC=3000)

4.2.4 Scalability Analysis of PDES

The next experiment shows the scalability of the simulator as the number of hardware threads per node is increased (Figure 31(a) with remote percentage of 20% and Figure 31(b) with remote percentage of 80%). In particular, the simulation is performed on 8 nodes, and each PE is mapped to one hardware thread. The total number of PEs used on the x-axis is increasing as we increase the number of hardware threads used on each node. In the case of 8 PEs on the x-axis, the three

versions have a similar performance with each other because they each have a single thread or a process on each node communicating through MPI. However, as the number of hardware threads per node is increased, ROSS-CMT is able to take advantage of the more efficient communication among threads on the same node avoiding the use of MPI, achieving over 2.8X speedup over ROSS-MPI for a 64 way simulation at 20% remote percentage (the improvement is a more modest 1.5X in the case of 80% remote percentage).

When message consolidation and infrequent polling are used, the optimized version of ROSS-CMT is able to scale much better, achieving about 4X improvement in performance over ROSS-MPI at both 20% and 80% remote event percentages. In addition, the performance of optimized ROSS-CMT exceeds that of baseline ROSS-CMT by a factor of 1.4X to 2X.

In the next experiment we show the scalability of the simulator as we increase the number of nodes used with a fixed number of hardware threads used per node (6 hardware threads). Figure 32(a) and Figure 32(b) show the scalability for optimistic simulation at 20% and 80% remote communication percentage respectively. We start with 2 nodes since it is impossible to create remote message traffic when only one node is used. Again, performance and scalability of the optimized ROSS-CMT are significantly better than the ROSS-MPI, achieving about 4X speedup at 80% remote message percentage and about 4.5X at 20% remote communication. In addition, in this experiment the performance of optimized ROSS-CMT exceeds that of baseline ROSS-CMT by a factor of up to 2X. Figure 33(a) and Figure 33(b) show the speedup of optimistic simulation against sequential simulation at both 20% and 80% remote communication percentages. Clearly, the optimized ROSS-CMT achieves better speedups than both baseline ROSS-CMT and ROSS-MPI. For example, at the case of 9 nodes, the speedup of optimized ROSS-CMT is 3.7X at 20% remote percentage, and 2.2 at 80% remote percentage.

In the next experiment we show the impact of Event Processing Granularity (EPC) for the scalability of PDES. EPC controls the amount of computation executed for each event. In our implementation the value of EPC determines the number of computation loops required for each event processing. A higher value of EPC makes the application more coarse-grained, and gives more computation load to the application. Note that most PDES models tend to require relatively small amounts of processing, typically to update state variables. However, it is possible to have models with significant event processing, and it is instructive to see how the

ratio of computation to communication influences the performance of PDES on CMs. Figure 34(a) and Figure 34(b) show the three versions of ROSS with EPC value of 3000 under 20% and 80% remote communication respectively. We find that the baseline ROSS-CMT performs close to optimized ROSS-CMT. This indicates that with a high value for EPC, the performance of ROSS-CMT is not dominated by communication.

In the next experiment, we evaluate PDES on a real model of a Personal Communication Services (PCS) system [11]. In this model, a cellular provider infrastructure is simulated as a number of mobile customers use it. A mobile phone call is simulated as an event, moved from one cell phone tower to another. Upon receiving a phone call, the cell phone tower assigns an available channel to the call. Once this phone call ends, the allocated channel is released. In addition, another phone call or more may be generated. If all channels are busy, the call is blocked. Moreover, if a call's connected portable is leaving the cell's area, then the call is handed-off to the destination cell phone tower [11].

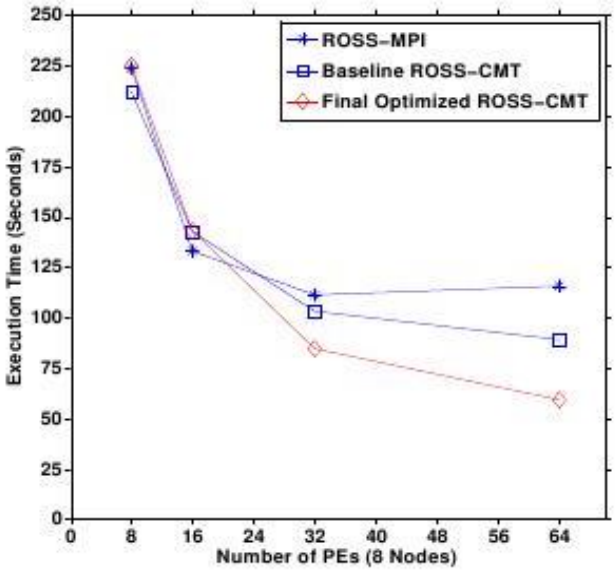


Figure 35 - Scalability Study of PCS Model

In our experiment, the PCS simulation consists of 65536 cell phone towers (LPs). In addition, the number of channels per cell phone tower is fixed at 10. Figure 54 shows the PCS scalability of three versions of ROSS simulator on 8 nodes. The simulation is performed under 8-way, 16- way, 32-way and 64-way, with 1 hardware thread, 2 hardware threads, 4 hardware threads and 8 hardware threads per node respectively. At the case of 64-way simulation, the

optimized ROSS- CMT exceeds the baseline ROSS-CMT by a factor of 1.5, and 2 over ROSS-MPI.

4.2.5 Experimental Results Summary

In summary, we evaluated the performance of ROSS-CMT with proposed optimizations on CMs with highly heterogeneous delays. We discovered that:

1. Message consolidation significantly improves the performance of ROSS-CMT PDES simulator. It provides a performance gain of about 3X in ROSS-CMT against ROSS-MPI at high percentage of remote communication. However, the overhead of unsuccessful probes can't be ignored, especially in the case of low percentage of remote communication.

2. The optimization of infrequent polling is capable of providing up to another 20% gain in performance of ROSS-CMT.

3. Our latency-sensitive partitioning strategy without any of above optimizations provides up to 44% performance improvement over latency-oblivious partitioning.

4. With the proposed optimizations the performance of the optimized ROSS-CMT exceeds that of non-optimized ROSS-CMT by a factor of about 2X, and that of ROSS-MPI by a factor of about 4.5X.

4.3 PDES Evaluation on the Tilera

In this section, we present performance evaluation of fully optimized ROSS-MT (multithreaded ROSS) and ROSS-MPI (MPI based ROSS) on Tilera. First, we discuss the evaluation environment, and the simulation benchmark that we use.

4.3.1 Experimental Setup and Benchmark

We used the basic Phold benchmark for the initial set of experiments, because it allows us to easily explore a wide range of application characteristics using configurable parameters such as percentage of remote communications, EPC and the number of objects per PE. We can also control event population by configuring the number of events generated during the initialization of each LP (referred to as start events). We use the above mentioned parameters for evaluating different aspects of scalability of our implementation. Later, to access the performance impact of

various partitioning strategies, we use a more restricted version of Phold, where non-uniform communication patterns are introduced to make the model sensitive to partitioning choices. This model is described later in this section.

4.3.2 ROSS-MT Scalability Analysis

Our first experiment was to study the scalability of the Phold model executed on Tilera as the number of cores used for simulation increases from one (sequential simulation) to 56 (the maximum number of cores available to us). For this experiment, we used 56000 total objects with equal number of objects per PE. We performed the experiments for three different values of remote communication percentage: 20%, 40% and 100%. The baseline for these experiments (the case with 1 core) represents the ROSS simulator running in the optimized sequential mode, without any of the overheads necessary for parallel simulation. The sequential ROSS simulator has an option of using a calendar queue or splay tree for the critical event queue [64]; we experimented with both data structures and selected the splay tree because it provided better performance. The sequential simulation run-time was 444 seconds with the splay tree, and 470 seconds with the calendar queue.

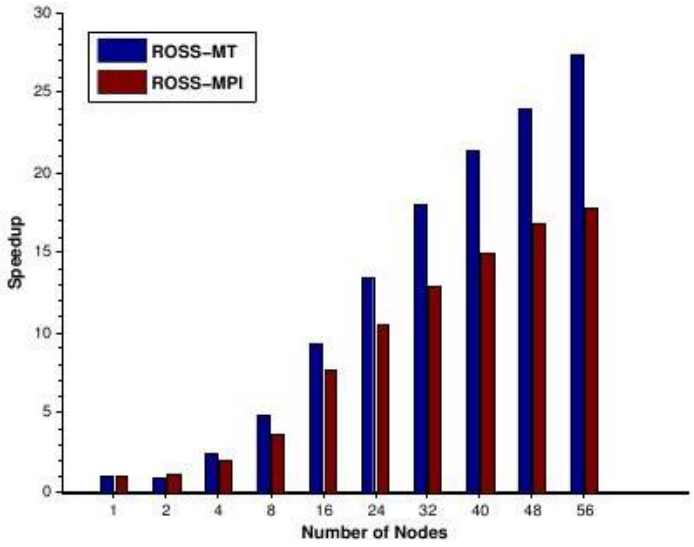


Figure 36 - Speedup at 20% Remote Communication

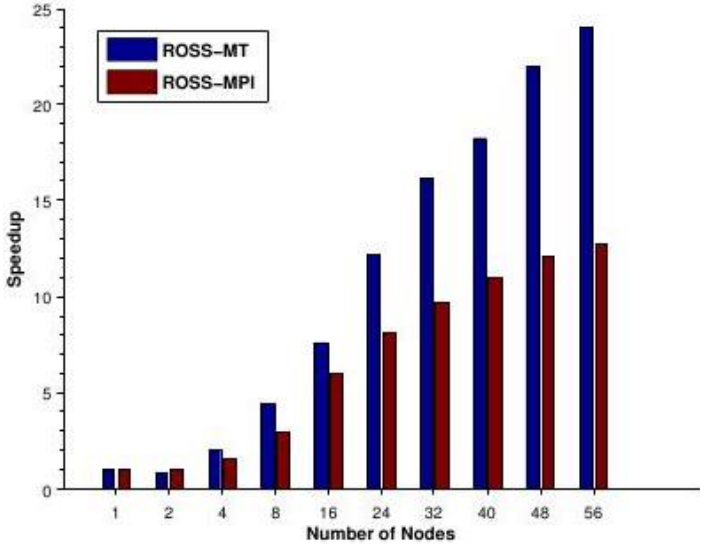


Figure 37 - Speedup at 40% Remote Communication

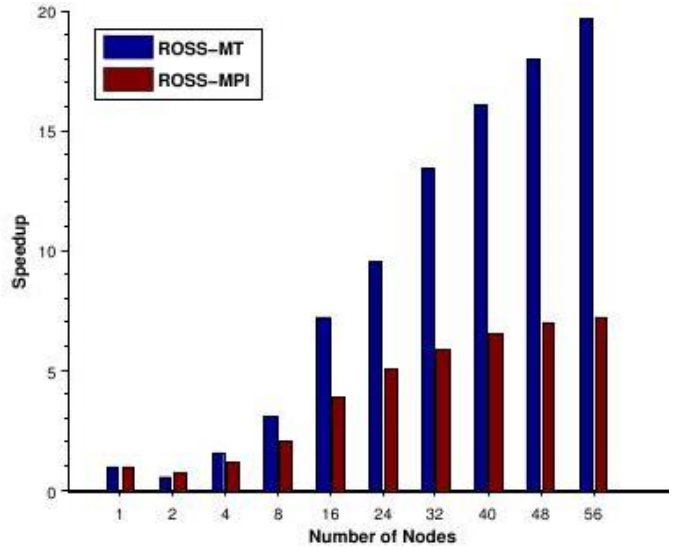


Figure 38 - Speedup at 100% Remote Communication

As shown in Figures 4.18, 4.19 and 4.20, ROSS-MT is significantly more scalable than ROSS-MPI. For example, for 20% remote events, ROSS-MT exhibits the speedup of 27X at 56-way parallelism, while ROSS-MPI shows the speedup of 18X at a similar setting. For 40% remote events, the respective speedups are 24X and 12X, and for 100% remote events the speedups are still significant, especially for ROSS-MT – 20X and 7X respectively. Note that ROSS-MT generally maintains better scalability trends than ROSS-MPI, the difference between the two increases as the percentage of remote events goes up. The main reason for better scalability on Tilera compared to the traditional multicore architectures is reduced lock contention overhead on Tilera due to the more efficient nature of the communication network.

Compared to the MPI implementation, ROSS-MT also saves processing cycles used for message probing in MPI based implementation.

Finally, we observed that the speedup achievable on Tilera is not constrained by the extra pressure on the communication network (and thus higher latencies), but instead is limited by the increased processing delays due to the software overhead of processing remote events. More results demonstrating this effect are presented later.

Next, we evaluate the impact of the performance optimizations for ROSS-MT (described in previous sections) executing on Tilera. These optimizations are driven by the observation that the scalability is limited by three major factors: barrier synchronization, NUMA issues and lock contention on the shared queue. In order to study the importance of each of these optimizations on the Tilera platform, we did a thorough analysis of each optimization. As discussed in previous sections, we also evaluated the role of PER_THREAD_HEAP optimization. Our analysis is performed for both conservative and optimistic simulation.

Figure 39 depicts the results for the optimistic simulation. Here, NUMA and barrier optimizations play a smaller role. Barrier synchronization implementation based on condition variables and pthread_mutex (as in the baseline ROSS-MT) scales reasonably well on Tilera due to lowered lock contention and a relatively low cost of remote cache access.

Similarly, the NUMA optimization plays a very important role on AMD and Intel platforms [34]. However, since on the Tilera chip all cores and the memory controller are tightly connected in a mesh, the effect of NUMA optimization is significantly smaller. Further, a low clock rate of Tilera cores reduces the mismatch between the CPU speed and the memory access time, thus diminishing the impact of non-uniformity in the memory access latency. Finally, the NUMA optimization also has a negative impact of increased fossil collection overhead. The combination of all these reasons makes NUMA optimization's impact on the overall simulation performance almost negligible.

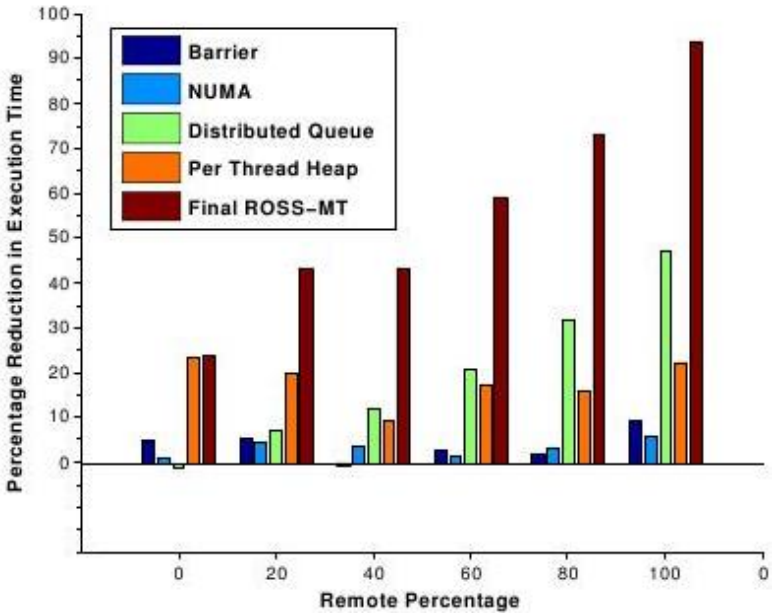


Figure 39 - Impact of Optimizations with Increasing Remote Percentage for Optimistic Simulation

On the other hand, the distributed queue optimization plays a major role in increasing the scalability and performance of baseline ROSS-MT, with up to 40% reduction in execution time. Our experiments show that 8 PEs can share a single queue without lock contention at 100% remote communication. This implies reduced lock contention as compared to the AMD Magny-Cours platform. Enabling the PER_THREAD_HEAP feature in the Tilera library reduces memory management overhead by reducing lock contention for a central heap. It also promotes local cache access by placing allocated pages on the same tile.

Figure 40 shows the performance benefits achieved by each optimization for conservative simulation. As shown in this figure, barrier and NUMA optimizations have a noticeable impact. Conservative simulation involves frequent synchronization of PEs, resulting in higher impact of barrier synchronization. Advantages of NUMA optimization come only from the LIFO strategy for conservative simulation. In addition, conservative simulation does not involve fossil collection and thus avoids the negative impact on NUMA-aware optimization. Because of the less frequent access to the input queue, the distributed queue based optimization plays a relatively small role in conservative simulation. At the same time, the use of PER_THREAD_HEAP shows significant improvement for both conservative and optimistic scenarios.

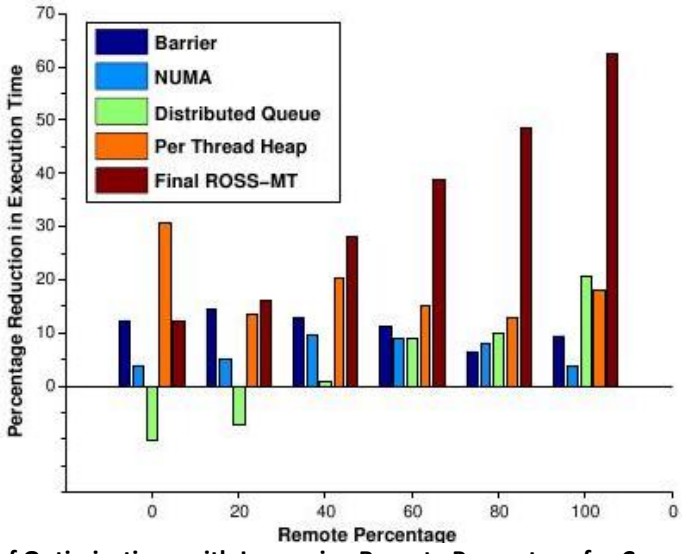


Figure 40 - Impact of Optimizations with Increasing Remote Percentage for Conservative Simulation

Next, we evaluate performance of ROSS-MT with increased remote communication. For this experiment, we use the basic Phold model in 56-way configuration with 1000 objects per PE. We fix the GVT interval at 256. Execution time is measured at different remote percentage for fixed batch size. We observed that the batch size of 8 is optimal for both multi-threaded ROSS and MPI-based ROSS. We set event processing factor to 0 and thus event processing overhead is limited to creating and sending a new event.

As shown in Figure 41, ROSS-MT significantly outperforms MPI-based ROSS. With the increase in remote communication, the execution time for ROSS-MPI grows linearly. At the same time, the execution time for ROSS-MT increases slightly with increasing remote communication. This behavior can be explained by the fact that additional processing delays introduced for handling remote communications at both sending and receiving nodes dominate the actual wire delays through the iMesh network. For ROSS-MPI, this software overhead of remote message processing and generation is much higher than for ROSS-MT, leading to differences in performance. When all communications are remote (100%), ROSS-MT outperforms ROSS-MPI by a factor of more than 3.

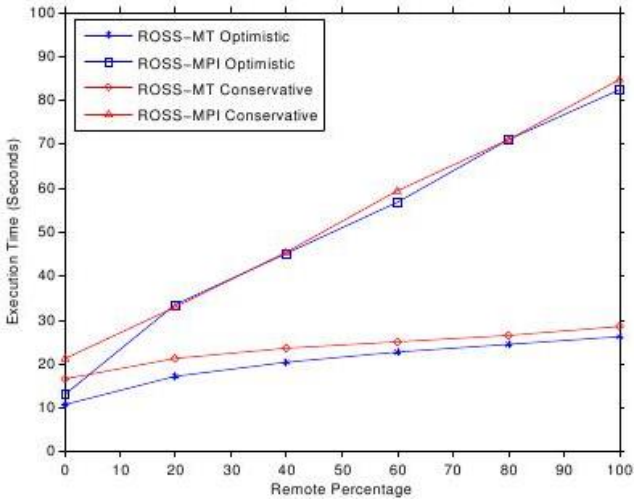


Figure 41 - Execution Time for ROSS-MT and ROSS-MPI on Tilera

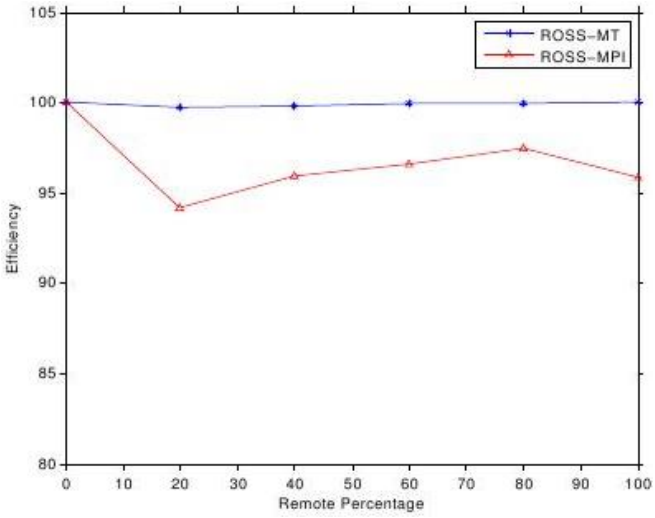


Figure 42 - Efficiency at GVT Interval 2048

In the next set of experiments, we analyze the impact of GVT interval on the event processing efficiency. For this study, we gradually increase the GVT interval from 256 to 2048. As shown in Figure 42, efficiency of the ROSS-MT stays noticeably higher than that of ROSS-MPI, even at relaxed GVT synchronization.

Another important observation specific to Tilera is that GVT computation cycle is relatively inexpensive due to the low-latency communication infrastructure. For the model such as basic Phold (that does not experience many rollbacks), this is manifested by the fact that the simulation performance remains relatively constant for different GVT intervals.

Thus, performing more frequent GVT computation cycles introduces negligible

overhead. Again, this example simply demonstrates that the GVT computation latency is low and computing GVT more frequently is likely to provide advantages especially for models with high rollback probability.

Next, we compare the performance of optimistic and conservative simulation on Tilera. For conservative simulation, the lookahead value is set to 1, while optimistic simulation generates events with time granularity of 1 (similar to lookahead). We set GVT interval to 2048 for this set of experiments. As shown in Figure 41, ROSS-MT outperforms ROSS-MPI by the same factor even for conservative simulation.

4.3.3 Stress-testing the iMesh

The iMesh tile interconnect provides low-latency and high-bandwidth communication among the tiles. The bandwidth of the iMesh interconnect is an important factor affecting the scalability of shared memory applications. Thus, it is important to evaluate ROSS-MT scalability with reference to the iMesh bandwidth. The next experiment that we present is our attempt to saturate the iMesh bandwidth by increasing event population in the Phold model. To this end, we used 56-way Phold simulation with 1000 objects per PE at 100% remote communication, and then gradually increased the event population by varying the number of starting events per LP. As shown in Figure 43, the event rate sustains even for 9 starting events per LP for 100% remote case. Instead of saturating the iMesh, the additional events simply exert the extra pressure on the processing cores, because more core cycles are needed to process the remote events.

The distributed shared cache feature of the Tilera platform plays an important role in determining the performance of highly parallel applications. Thus, it is essential to study the scalability of ROSS-MT with increasing cache pressure by increasing per-tile memory demand. In this experiment, we evaluate the performance of ROSS-MT by gradually increasing the number of LPs per PE to a very high number. As shown in Figure 44 event rate sustains even for 50000 LPs per PE at 100% remote communication. Thus ROSS-MT can scale very well on the Tilera for very large PDES models; performance is only constrained by the computational power of the individual cores.

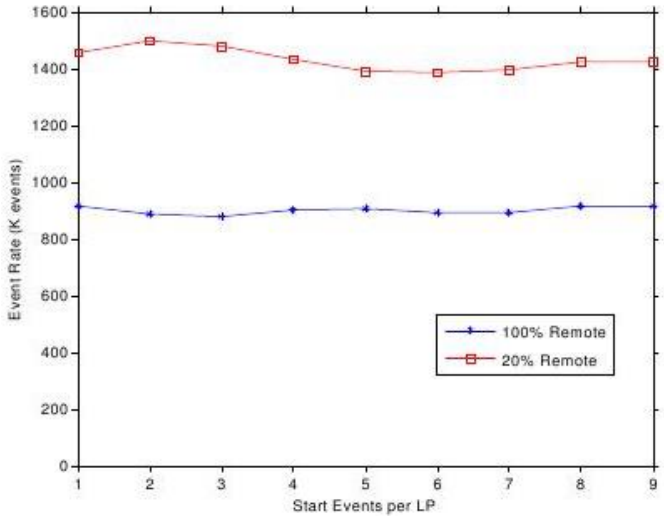


Figure 43 - ROSS-MT Performance with Increasing Event Population

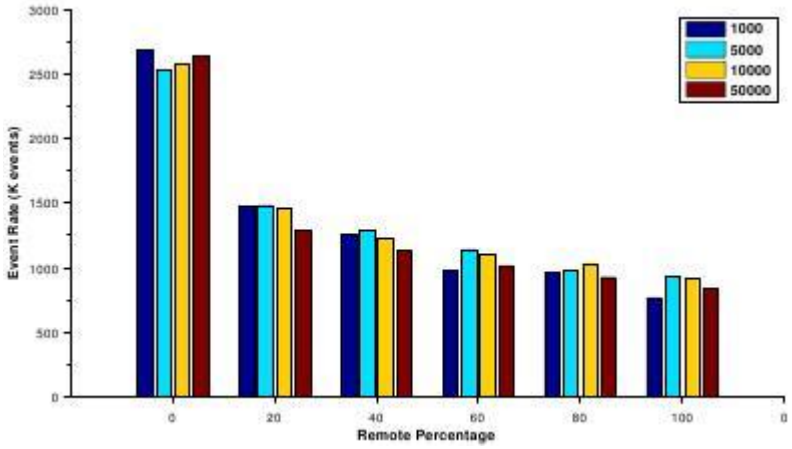


Figure 44 - Scalability with Increasing LPs per PE at Different Remote Percentages

4.3.4 Partitioning and Placement Issues

Our previous results reported in this section (especially the attempt to saturate the iMesh network) suggest that communication plays a secondary role in defining PDES performance on the Tilera and that the real bottlenecks are the processing cores themselves. We now project this vision into the issue of model partitioning and object placement, which in traditional architectures play an important role for PDES performance. In fact, complex partitioning schemes that accurately balance communication and computation often provide serious performance gains on those systems. In addition careful object placement is also a must.

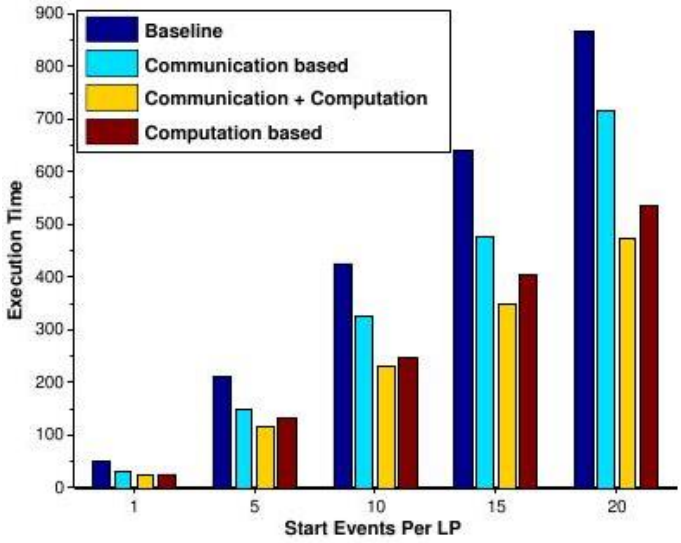


Figure 45 - Performance Comparison of Various Partitioning Strategies

To understand the impact of model partitioning while executing PDES on the Tilera, we augmented the Phold model with the capability to define event processing granularity (EPC) for each event. We also overlaid a hierarchical topology on top of the objects such that each object communicates to a fixed set of other objects to make the model sensitive to partitioning (simple random Phold is not). With this enhanced model, we experimented with partitioning schemes that: a) emphasize reduction in the number of inter-core events over load balancing, b) balance computational load without considering the impact on communication; c) equal emphasis on both; and d) randomly distributes objects among the cores to act as a baseline for comparison purposes.

As shown in Figure 45, while the partitioning strategy that takes into account both computation and communication achieves the best performance, the partitioning strategy that emphasizes the computation balance achieves comparable level of performance (within 10% in all cases considered). This result corroborates our previous hypothesis about the computation-dominated nature of the Tilera. The significance of this result is that even a simple partitioning scheme that just balances computation is sufficient to sustain PDES scalability on the Tilera. This kind of partitioning is much easier to obtain than a one that optimizes for communication. For example, the information about event processing is readily available from the model, while communication frequencies have to be obtained through profiling. Furthermore, complex graph partitioning tools are needed to derive communication-optimized partitions. Finally, the same

observations apply to dynamic object migration schemes. It is much easier to design them if only the information about the CPU loading needs to be maintained.

In our next experiment, we evaluate the impact of placement of highly communicating PEs on the performance of ROSS-MT. Due to the mesh topology of tile interconnect, it is logical to place highly communicating PEs on the adjacent tiles for higher performance. In order to evaluate this, we modified the basic Phold, so that communicating pairs of consecutive PEs are formed e.g. (0,1),(2,3). LPs in one PE communicate only with other LPs in the same pair. We measure the execution time of simulation by two PE placement strategies: Nearby and Random. In Nearby placement strategy, we place PEs that form a pair on physically adjacent tiles, while in Random placement, PEs in one pair are placed on physically farthest tiles.

Table 11 - Placement of Paired Model

Remote %	Nearby placement	Random placement
0	8.00	8.03
20	12.18	12.44
40	14.09	16.64
60	14.96	17.18
80	18.57	19.07
100	21.34	21.98

As shown in Table 11, such placement variations have a negligible impact on the performance, confirming once again that the iMesh is not a bottleneck and that object placement should be a secondary consideration while running PDES on the Tilera.

4.4 Evaluating PDES Partitioning Schemes

In this section we present a number of experiments to evaluate the impact of dynamic partitioning on simulation performance.

4.4.1 Impact of Topology

In the first experiment, we show the performance of dynamic behavior based partitioning for different activity patterns overlaid on top of the two different static topologies. Figure 46 shows the execution times with Static and Activity partitioning strategies for Hierarchical and Uniform models with all six frequency distributions (Refer to Figure 10). The results show that, in the case

of Hierarchical model, for the first four Alpha values static connectivity based partition performs equally well as the activity partitioning. In these cases, activity patterns match the underlying structure making Static partitioning effective. However, for last two Alpha values activity differs significantly from the underlying connectivity. This information is captured and utilized by Activity partition and results in 20 to 40% benefit over Static partitioning.

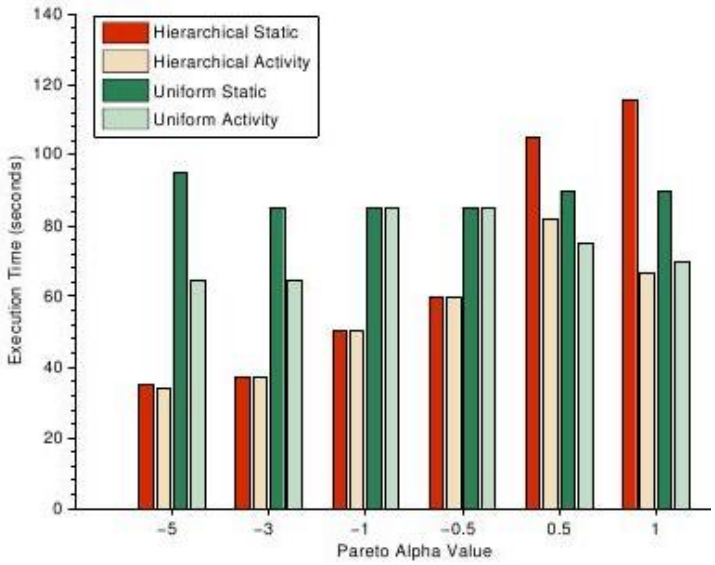


Figure 46 – Hierarchical Vs. Uniform (Cluster 8 nodes)

Partitioning for the Uniform model reveals more about the connectivity-activity interplay. Since the Uniform model does not have a partitionable static connectivity pattern, the activity patterns influence the model behavior even in case of smaller Alpha values; the emergent dynamic pattern can be identified and used to partition the model. Thus, the effectiveness of partitioning is a function of the dynamic behavior of the model, regardless of the static relationships.

In the next experiment, we study the impact of dynamic partitioning on a Hierarchical fine-granularity, communication intensive application. Figure 47(a) shows the execution times for three partitioning strategies: Random, Static and Activity. For this scenario, activity-based partitioning substantially outperforms static partitioning by a factor of 2x on 4 nodes and 4x on 32 nodes. To explain this performance, Figures 4.29(b) and 4.29(c) show the static and dynamic mincut achieved by the partitions. The mincut represents the cut size once the graph is partitioned. The static cutsize considers all edges of equal weight (1). In contrast, the Active cutsize takes into account the frequency of communication on each edge. While static partitioning achieves a better static mincut, Activity partitioning achieves substantially better dynamic mincut size. In effect, the

number of communicated messages reflected by the Active mincut, is greatly reduced, resulting in the better performance achieved in comparison to static partitioning.

In Figure 48(a) we repeat the experiment for a Hierarchical, computation intensive model where event execution requires substantial time (simulated using a delay loop). For high granularity models, Activity partitioning does not clearly outperform Static partitioning. Since the simulation time is dominated by event processing, the primary consideration for effective processing is load balancing. Figures 4.30(b) and 4.30(c) show that the Static and Active mincuts of Activity and Static partitions closely follow each other.

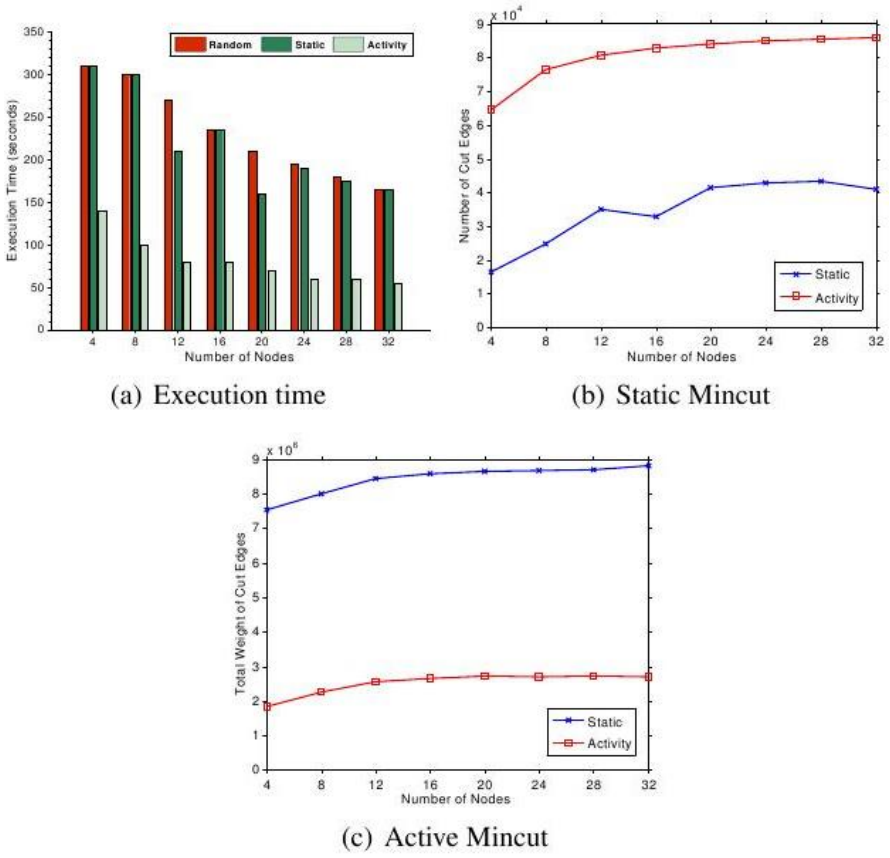


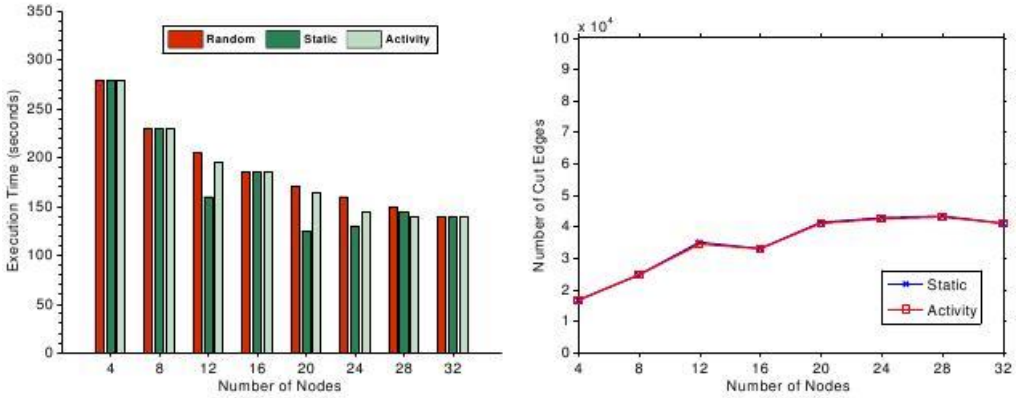
Figure 47 - Communication Intensive Model

In Table 12, we show the dynamic load balancing achieved by the activity based partitioning for the fine-granularity model. The activity based partitioning only considers the dynamic communication information, but only load balances in terms of the object counts. Since object behavior also varies significantly, we can see in the table that it results in large deviations in load balance in terms of total object weights. We need to take into account dynamic resource

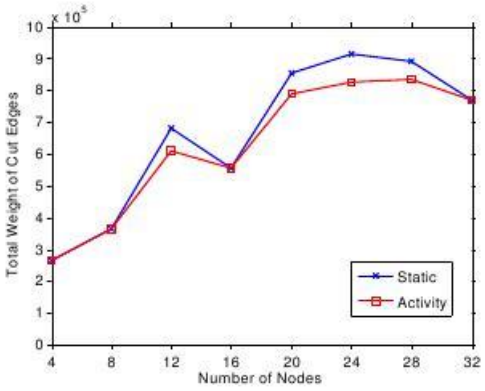
usage of the objects.

Table 12 - Communication Intensive Model: Object Deviation

Nodes	4	8	12	16	20	24	28	32
Static Activity	363 87708	333 55720	23484 50401	169 68579	16499 48606	16034 45604	14907 40209	51 41287



(a) Execution Time (b) Static Mincut



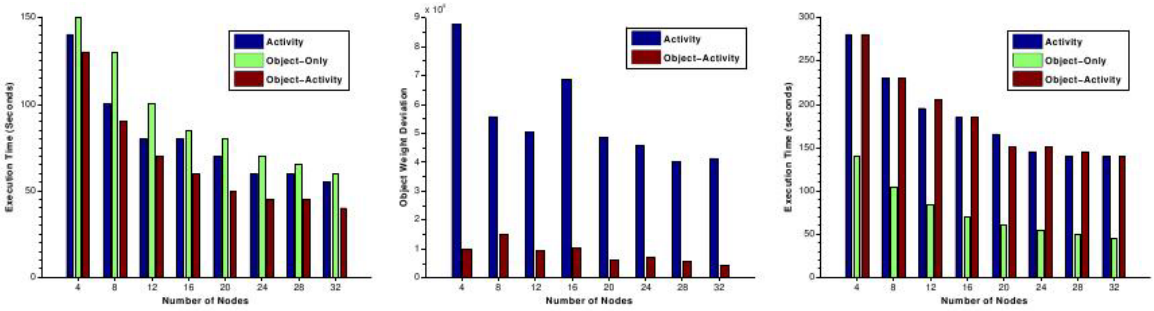
(c) Active Mincut

Figure 48 - Computation Intensive Model

4.4.2 Combined Communication and Load Balancing

Table 12 indicated that there is further opportunity in case of Activity partition as it does not consider load balancing. Object-Activity partitioning strategy is useful where load balancing is important but we also need to consider communication. Figure 49(a) shows around 25 to 30% benefit of Object-Activity partition over basic Activity partition in a communication intensive model. Figure 49(b) shows the reduction in object deviation in the case of Object-Activity partition. Object-Only partition, which solely focuses on load balancing, does not perform well in this case

as it fails to consider the communication.



(a) Object-Activity: Communication aware Load Balancing (b) Object-Activity: Object Deviation (c) Object-Only: Computation Focused Load Balancing

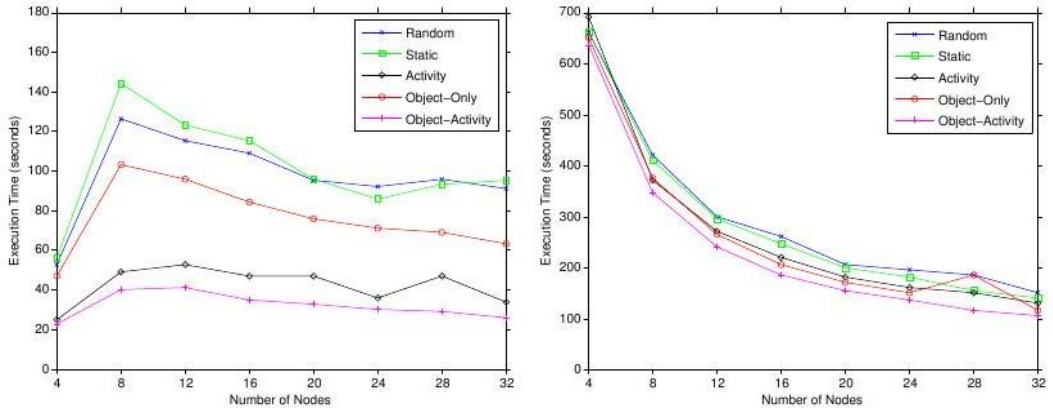
Figure 49 - Importance of Load Balancing

Figure 49(c) shows the impact of Object-Only and Object-Activity partitions in the case of a computation intensive model. In this model, even though communication patterns exist, computation plays a more important role in determining overall performance. This is reflected in up to 3x benefit of Object-Only partition over Activity and Object-Activity partition. Object-Activity partition is primarily an Activity partition with a load balancing component. However in this case, where the communication is not significant, the Object-Only partition which purely focuses on load balancing performs better.

4.4.3 Effect of Processing Granularity

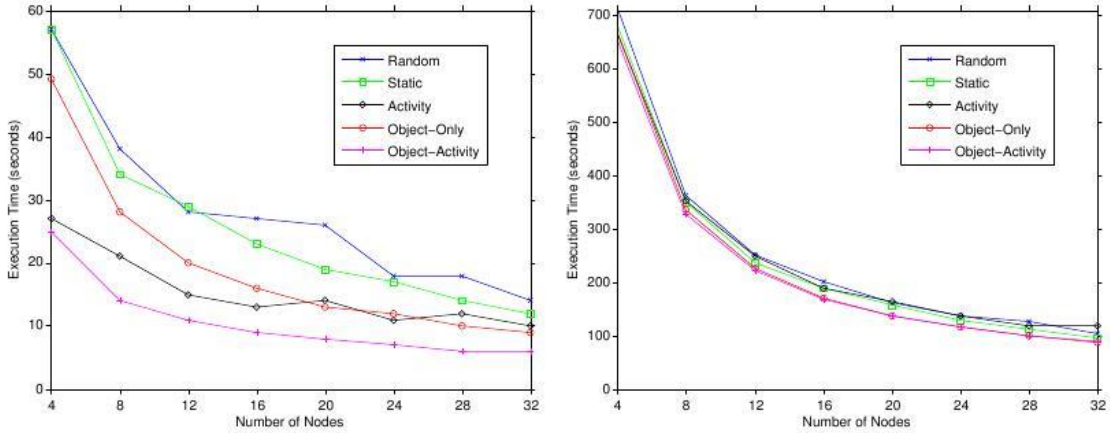
Figures 4.32 and 4.33 show the comparison of a communication-bound and computation-bound model on a cluster of multi-core machines and an equivalent configuration on the AMD Magny-Cours respectively. Figures 4.32(a) and 4.33(a) show 4-way to 32-way execution of a communication-bound model (Pareto Alpha 1). Despite having slightly slower cores, the AMD machine was able to achieve significantly better performance than the cluster. This advantage can be explained by the low communication costs on a multi-core machine, in comparison with the high cost of network communication. The second observation is that the Object-Activity partition gives significant benefit on the cluster but not nearly as much on the multi-core machine where the low latency places a premium on load balanced partitions. Figures 4.32(b) and 4.33(b) show a computation bound model with a large, 3000 iteration delay loop for each event. As expected, the execution times are much higher. In addition, execution times on AMD Magny-Cours are now only 10 to 15% better than those on cluster. In this case, the communication properties of the

model were the same as the communication-bound model used here. Therefore this trend indicates that the computation intensive event processing had much higher impact on the Magny-Cours machine than on the cluster.



(a) Communication-bound Model (b) Computation-bound Model

Figure 50 - Cluster Performance



(a) Communication-bound Model (b) Computation-bound Model

Figure 51 - Performance on an AMD Magny-Cours 48-core Machine

4.5 Evaluating Interference-Resilient PDES

This section presents a performance evaluation of ADM and LADM in comparison to FM. Most of the experiments use the Phold benchmark [26], with 1000 objects per PE. For some experiments, we use Personal Communication Services (PCS) system to show that the technique transitions to real models [11].

4.5.1 Evaluation of ADM

In the first experiment, we evaluate the performance of ADM without the data locality optimization. Figure 14 and Figure 15 show the performance of ADM compared to FM and baseline DM on the Core i7 and Magny-Cours platforms respectively. In this experiment, we introduce the interfering load at the start of the simulation, and it runs for the duration. In Figure 14(a) and Figure 15(a), we see the execution time as a function of the percentage of remote communication (the communication between PEs). ADM achieves better performance than FM on the Core i7, but only outperforms FM at high remote communication (≥20%) percentages.

The behavior can be partially explained by the high cost of lower level cache accesses on the Magny-Cours relative to the Core i7. At 100% remote communication, for example, ADM can achieve a speedup of 2.8X against FM on the Core i7 machine, and 1.8X on the Magny-Cours platform. ADM performs closer to the ideal runtime (predicted by proportional slowdown) on the Core i7 platform than on the Magny-Cours machine. Thus, the benefit from ADM is offset somewhat by the increased cache misses and the poorer locality that results from continuous re-mapping of the work. The locality aware version of ADM attempts to address this issue. As described previously, the baseline DM will experience poor efficiency when contention becomes heavy, while ADM can reduce such contention by deactivating some threads. To demonstrate this, we present efficiency of corresponding simulations (as shown in Figure 14(b) and Figure 15(b)). Clearly, the baseline DM exhibits a poor efficiency similar with FM on the Core i7 machine, but can achieve relatively better efficiency on the Magny-Cours machine. In contrast, ADM can achieve efficiency of over 90% on both platforms.

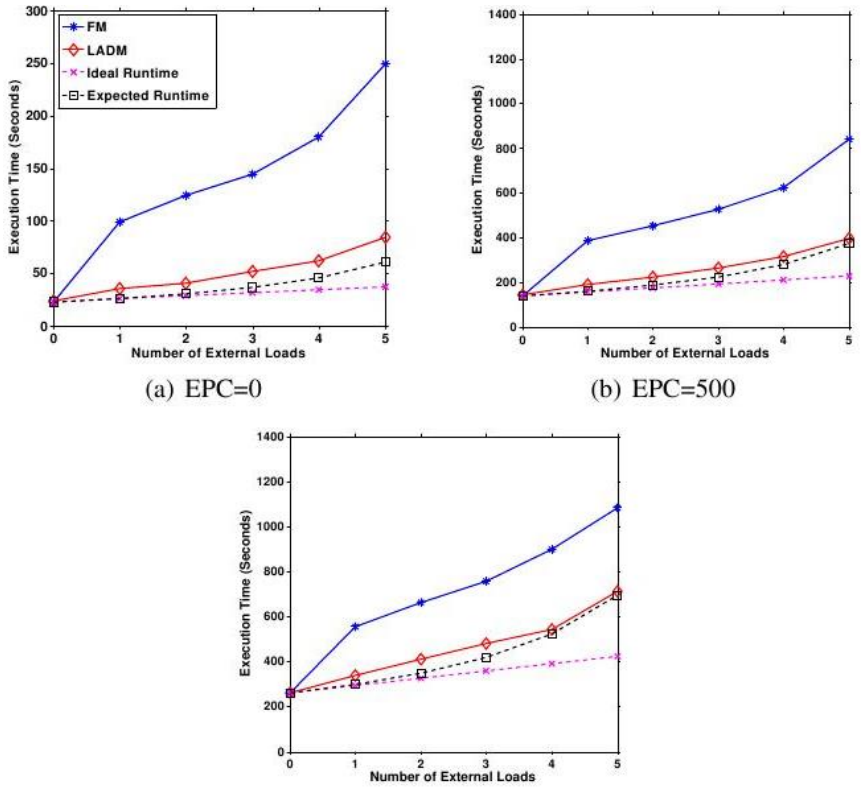


Figure 52 - Impact of Event Processing Granularity on the Intel Core i7 Machine

4.5.2 The Impact of Data Locality

In ADM, each active thread moves to the next free PE in a round-robin fashion at the beginning of the simulation loop, even when there is no interference. Thus, ADM can lead to poor cache locality, as each thread accesses different PEs causing their state to be interrogated between caches. LADM improves data locality by associating threads with primary PEs. Only orphan PEs (those whose primary thread is inactive) experience a loss of locality as their events are processed by the other active threads.

The next experiment evaluates the performance of FM, ADM and LADM in the absence of external loads to measure the overhead of the mechanisms when they are not needed. As seen in Figure 16, LADM performs up to 11% better than ADM on the Core i7 machine, and up to 53% on the Magny-Cours machine. In addition, LADM incurs small performance loss (less than 5%) relative to the FM version. The overhead is partially due to the extra checking that LADM does; however, we also noticed that rarely, LADM incorrectly detects the presence of interference. We believe that there is room for improving the interference detection algorithm in future work.

In the next experiment, we consider a scenario with 1 external interfering process (Figure

17). At high remote communication (≥20%), LADM outperforms the original FM by a factor of up to 2.8X on the Core i7 machine, and up to 2.2X on the Magny-Cours machine. In addition, LADM performs up to 43% better than ADM on the Magny-Cours machine, due to the fact that LADM can achieve better data locality. Figure 18 shows the cache miss rates, demonstrating how LADM has substantially lower cache miss rates than ADM.

4.5.3 Impact of Event Processing Granularity

In the next experiment we modify the Phold model to increase the granularity of event processing time. In particular, the granularity is controlled using shows the effect of event processing granularity on the performance of optimistic PDES in the presence of interference. A new parameter, called EPC, is defined to control the amount of computation for each event processing in Phold. A higher value of EPC gives more computation load increasing the ratio of computation to communication.

We evaluate the performance of FM and LADM as the number of external loads is increased, for both the Intel (Figure 52) and AMD Magny-Cours (Figure 53) platforms at remote communication percentage of 40%. As seen in Figure 52 and Figure 53, LADM performs better than FM on both platforms when the simulation is interfered by external loads. In addition, the gap with the ideal performance is decreased as EPC increases.

Moreover, the gap between LADM and the ideal performance on the Magny-Cours machine is larger than that on the Core i7 machine. We believe that the penalty of a cache miss on the AMD Magny-Cours machine is high, due to its NUMA characteristics. Another interesting observation is that FM performs closer to LADM as EPC increases. We discover that FM is capable of achieving relatively better efficiency at higher EPCs. As each event requires more time for processing in the case of higher EPC, the advance rate of each PE in FM is more balanced than that in the case of lower EPC.

4.5.4 Performance Evaluation of PCS Model

In this experiment, we study a model of a Personal Communication Services (PCS) system [11]. The PCS model simulates how a cellular provider infrastructure handles a number of mobile phone calls. In this model, an event represents a mobile phone call, sent from one cell phone tower to another. Each cell phone tower has a fixed number of channels. Upon receiving a call, the cell

phone tower assigns an available channel to the call, and later releases the allocated channel when the call completes. If all channels are busy, the call is blocked. In addition, the call is handed-off to the destination cell phone tower if the call's connected portable is leaving the area of the cell phone tower [11, 10].

The PCS simulation consists of 36864 cells (LPs) distributed among 8 PEs on the Intel Core i7 machine, and 48 PEs on the AMD Magny-Cours machine. Moreover, we fixed the number of channels per cell phone tower at 10. Figure 54(a) and Figure 54(b) show the performance of PCS model in the presence of external loads on the Core i7 machine and the AMD Magny-Cours machine respectively. Clearly, LADM performs better than FM on both Platforms. At the case of 5 external loads, for example, the performance of LADM exceeds that of FM by a factor of 3.7X on the Core i7 machine. In addition, LADM outperforms FM by a factor of about 2.5X at the case of 30 external loads on the Magny-Cours machine.

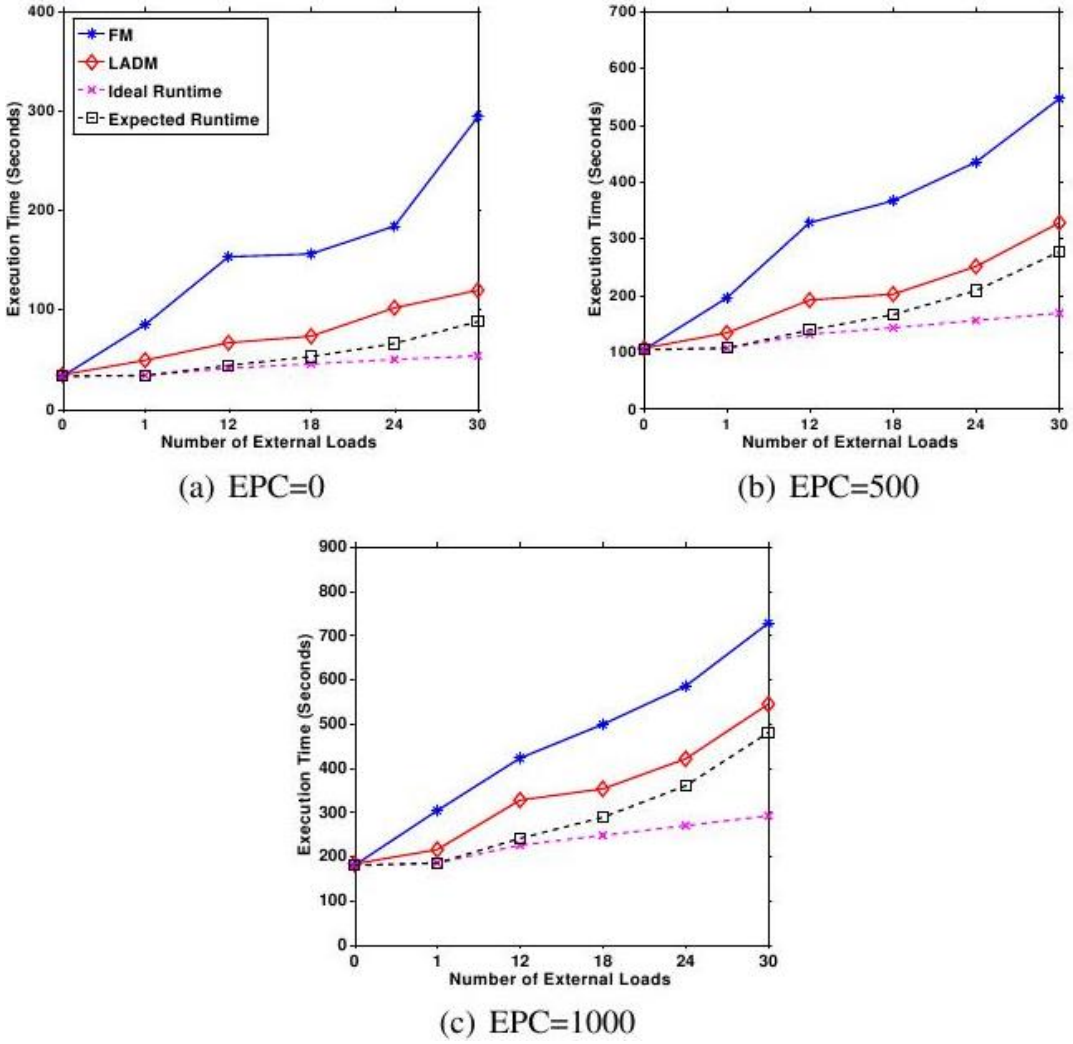
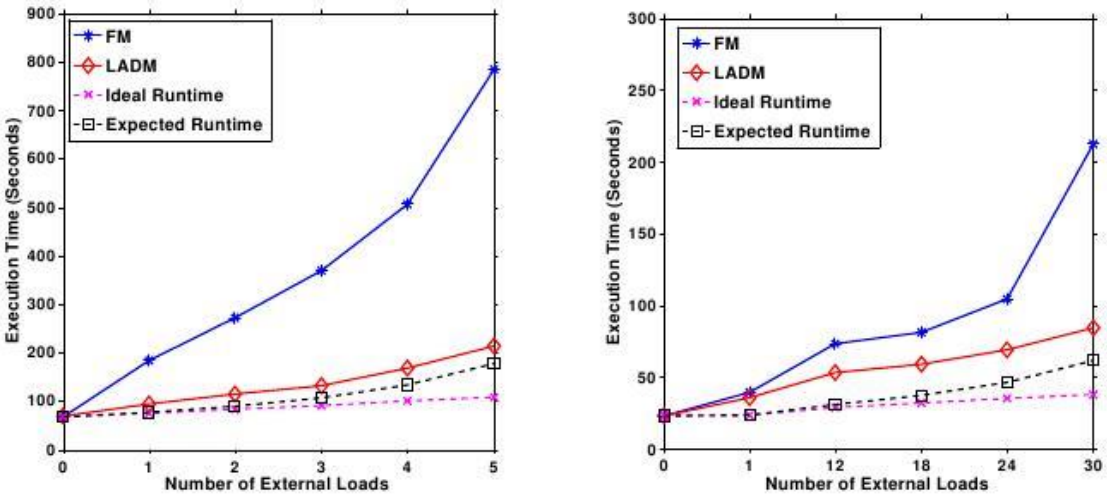


Figure 53 - Impact of Event Processing Granularity on the AMD Magny-Cours System



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 54 - Performance of PCS Model

5.0 Conclusions

This report summarizes the main accomplishments of a 2-year project targeted at developing scalable algorithms and techniques for achieving high performance and resilient execution of PDES on multicore machines and their clusters. Our studies proceeded along several major directions.

First, we presented our experiences in building a multi-threaded PDES simulator optimized to representative state-of-the-art multi-core machines. We used the ROSS PDES simulator, and reimplemented it from a process based model to a thread based model. Performance evaluation of the base implementation showed significant performance benefits on Core i7, but surprisingly poor performance on the AMD Magny-Cours.

We studied the reasons for this poor performance, and identified three bottlenecks. First, the barrier and all-reduce primitives used in GVT computation were implemented in an inefficient way using condition locks and broadcasts. We replaced it with an implementation that is based on pthread_barrier, which uses native machine instructions for implementing barriers, significantly improving the performance of GVT. The second problem was due to free memory management, which was not sensitive to the NUMA nature of the Magny-Cours platform. We addressed this problem by splitting the free memory pool to keep track of the memory origin for future allocation. The third bottleneck was due to the lock contention on the input queue. We resolved this issue by splitting the queues to reduce contention from all threads to only two threads for each queue.

The optimizations resulted in substantial improvement in performance; optimized ROSS-MT outperforms the MPI version by a factor of up to 3 on Core i7 and up to 1.2 on Magny-Cours. We also explore a preliminary implementation of a no-copy version of our communication primitives, which significantly improves performance especially when message lengths are large.

Then, we attempted to answer the following question: in the presence of highly heterogeneous delays on clusters of multicore machines, do fine-grained applications such as PDES, benefit from the low latency between cores on a single machine, or are they limited by the communication across machines with relatively higher delay? We illustrated this problem by using ROSS-CMT PDES simulator. We found that on CMs, for ROSS-CMT, the network connections impact its performance and scalability significantly.

We proposed three optimizations to reduce the impact of communication across machines: consolidated message routing, infrequent polling, and partitioning aware of the heterogeneous latency. We discover that the performance of optimized ROSS-CMT with classical Phold model achieves a 2X speedup against non-optimized ROSS-CMT, and 4.5X speedup against the original multi-process ROSS simulator. Latency-sensitive partitioning can provide up to 44% performance improvement against latency-oblivious partitioning on CMs.

Next, we presented detailed characterization and evaluation of PDES on Tilera Tile64Pro architecture - an example of emerging class of many-core designs. In contrast to traditional communication-dominated parallel computing platforms (for which classical PDES algorithms and techniques were designed and optimized), Tilera represents a computation-dominated environment which has significant implications on PDES.

Specifically, we demonstrated that large speedups can be achieved on Tilera, especially when designs are optimized to take into account the presence of on-chip shared memory hierarchy (as in ROSS-MT). We also demonstrated that PDES optimizations designed for traditional Intel and AMD chips do not necessarily work as such on Tilera and need to be adjusted to take into account the new computation-communication balance. Furthermore, we demonstrated that simple partitioning and dynamic object migration schemes that just take into account the computational balance across the cores provide performance that is competitive (within 10%) of more complex schemes that also optimize for communication. Next, we showed that the object placement across the cores should be a secondary consideration on Tilera, as there is nearly no performance difference between various placement schemes for the models that we considered. Finally, we showed that GVT calculations on Tilera are inexpensive, meaning that frequent GVT cycles can provide optimal performance for optimistic simulation, especially for rollback-prone models.

We then presented a case for partitioning based on dynamic model behavior. Most existing partitioning schemes only consider the static structure of the model. When application behavior exhibits large divergence from this static model, these approaches cannot effectively reduce communication or load balance the simulation. We showed that by taking the dynamic information into account, much more effective partitioning can result. In environments with high communication latency, this leads to up to 4x improvement in run-time for some

applications. Up to 2x improvement was observed in a low-latency multi-core platform. Our future work targets effective approaches for identifying the dynamic model behavior. In the current study, we collected this information through profiling. However, it is likely that static analysis, perhaps augmented with limited profiling, can provide a more attractive approach for estimating the model behavior.

Finally, we demonstrated the sometimes dramatic slowdown that can result in the presence of external interference. We presented a new metric, called proportional slowdown, to measure the idealized slowdown of PDES in the presence of interference and showed that in practice the observed slowdowns far exceed it. We proposed to use dynamic mapping to allow active threads to work on the PEs in a fair way, allowing the simulation to continue to proceed even if one or more threads are context switched. We then proposed a locality-aware dynamic-mapping (LADM) scheme that improves the locality of the proposed adaptive scheme by attempting to keep PEs assigned to their primary threads. Our experimental results showed that LADM is significantly better able to tolerate interference than fixed-mapping implementation, thus reducing the gap with proportional slowdown.

Our future work targets effective approaches for improving the algorithm of scheduling between PEs and threads. In the current study, each active thread looks up a PE on the orphan list in a round-robin fashion. We believe that a better locality can be achieved if the workloads of PEs on the orphan list are equally divided, and each workload is assigned to a specific active thread. We also plan to improve the accuracy of the interference detection algorithm.

This project resulted in a number of publications and multiple submissions that are currently under review, including publications in IPDPS, PADS Workshop and SIGSIM PADS conference – all premier venues in the field of parallel simulation and parallel and distributed computing in general.

6. 0 References

[1] G. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison Wesley, 2000.

[2] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Partitioning on dynamic behavior for parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 221–230. IEEE, 2012.

[3] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Partitioning on dynamic behavior for parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 221–230. IEEE, 2012.

[4] V. Balakrishnan, P. Frey, N. Abu-Ghazaleh, and P. Wilsey. A framework for performance analysis of parallel discrete event simulators. In *Proc. of the Winter Simulation Symposium*, 1997.

[5] D. Bauer, C. Carothers, and A. Holder. Scalable time warp on bluegene supercomputer. In *Proc. of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2009.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – a Gigabit-per-second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] A. Boukerche and S. Das. Dynamic load balancing strategies for conservative parallel simulation. In *Proc. of 11th Workshop on Parallel and Distributed Simulation (PADS)*, 1997.

[8] Azzedine Boukerche and Carl Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. In *Proc. of the eighth workshop on Parallel and distributed simulation (PADS)*, pages 164–172, 1994.

[9] C. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low memory, modular time warp system. In *Proc of the 11th Workshop on Parallel and Distributed Simulation (PADS)*, 2000.

[10] C. D. Carothers and R. M. Fujimoto. Background execution of time warp programs. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 12–19. IEEE, 1996.

[11] C. D. Carothers, R. M. Fujimoto, and Y-B. Lin. A case study in simulating pcs networks using time warp. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 87–94. IEEE, 1995.

[12] C.D.Carothers and R. M. Fujimoto. Efficient execution of time warp programs on heterogeneous, now platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11:299–317, 2000.

[13] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *IEEE International Paralleland Distributed Processing Symposium (IPDPS)*, 2010.

[14] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. Optimizing communication in Time-Warp simulators. In *12th Workshop on Parallel and Distributed Simulation*, pages 64–71. Society for Computer Simulation, May 1998.

[15] R. Child and P. Wilsey. Dynamically adjusting core frequencies to accelerate time warp simulations in many-core processors. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 35–43. IEEE, 2012.

[16] J. Cloutier. Model partitioning and the performance of distributed timewarp simulation of logic circuits. *Simulation Practice and Theory*, (1):83–99, 1997.

[17] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[18] L. Dagum and R. Menon. OpenMP: an industry standard API for shared memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[19] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, editors, *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, December

1994.

[20] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore arhcitectures. In *Proc. Of ACM/IEEE Confernece on Supercomputing*, 2008.

[21] R. Eduardo, D. Grande, and A. Boukerche. Dynamic load redistribution based on migration latency analysis for distributed virtual simulations. In *Haptic Audio Visual Environments and Games (HAVE)*. IEEE, 2011.

[22] R Ewald, C Maus, A Rolfs, and A Uhrmacher. Discrete event modelling and simulation in systems biology. *Journal of Simulation*, 1(2):81–96, May 2007.

[23] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *Proc. WSC*, pages 848–855, 2006.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[25] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[26] R. Fujimoto. Performance of time warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):23–28, January 1990.

[27] R. Fujimoto and K. Panesar. Buffer management in shared-memory Time Warp system. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pages 149–156, June 1995.

[28] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, 1997.

[29] R. M. Fujimoto, J. Tsai, and G. C. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

[30] R. Govindan and Hi Tangmunarkunkit. Heuristics for internet map discovery. In *Proc. of Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2000.

[31] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[32] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proc. ASPLOS*, pages 54–63, 1989.

[33] A. Holder and C. Carothers. Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer. In *European Modeling and Simulation Symposium*, 2008.

[34] D. Jagtap, N.Abu-Ghazaleh, and D.Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proc. of IPDPS*, 2012. (forthcoming).

[35] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[36] G. Karypis and V. Kumar. hmetis: a hypergraph partitioning package. Available on WWW at URL: http://www.cs.umn.edu/ karypis/metis/hmetis.

[37] K. Kim, T. Kim, and K. Park. *Journal of Systems Architecture*, 44(6-7):433–455, March 1998.

[38] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, 2002.

[39] Pavlos Konas and Pen-Chung Yew. Parallel discrete event simulation on shared-memory multiprocessors. In *Proc. of the 24th annual symposium on Simulation,* ANSS '91, pages 134–148, 1991.

[40] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the internet's router-level topology. In *Proc. of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 2004.

[41] L. Li and C. Tropper. A design-driven partitioning algorithm for distributed verilog simulation. In *Proc. 20th International Workshop on Principles of Advanced and Distributed*

Simulation (PADS), pages 211–218, 2007.

[42] Lijun Li and Carl Tropper. A design-driven partitioning algorithm for distributed Verilog simulation. In *Proc. of 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 211–218, 2007.

[43] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *Proc. Of ACM/IEEE conference on Supercomputing*, pages 58–71, November 2003.

[44] M. Low. Dynamic load-balancing for BSP time warp. In *Proceedings of the Annual Simulation Symposium*, pages 267–274, 2002.

[45] A. W. Malik, A.J.Park, and R.M. Fujimoto. Optimistic synchronization of parallel simulations in cloud computing environments. In *Proceedings of the International Conference on Cloud Computing*, pages 49–56. IEEE, 2009.

[46] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[47] Collin McCurdy and Jeffrey Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems & Software (ISPASS)*, 2010 IEEE International Symposium, pages 87 – 96, March 2010.

[48] M.D. McDowall, M.S. Scott, and G.J. Barton. Pips: human protein–protein interaction prediction database. *Nucleic acids research*, 37(suppl 1):D651–D656, 2009.

[49] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *Proc. of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2001.

[50] A. Medina, I. Matta, and J. Byers. On the origin of power laws in internet topologies. *ACM SIGCOMM Computer Communications Review*, 30(2), 2000.

[51] Biswajit Nandy and Wayne M. Loucks. On a parallel partitioning technique for use with

conservative parallel simulation. *SIGSIM Simul. Dig.*, pages 43–51, July 1993.

[52] A. Nataraj, A. Morris, A. Malony, M. Sottile, and P. Beckman. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proc. of ACM/IEEE Confernece on Supercomputing*, pages 1–12. IEEE, 2007.

[53] R. Noronha and N. B. Abu-Ghazaleh. Active nic optimization for time warp. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[54] J. Owens, W. Dally, D. Jayasimha, S. Keckler, and L. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[55] K. Perumalla. Scaling time warp-based discrete event execution to 1024 processors on a blue gene supercomputer. In *Proc. Computing frontiers*, pages 69–76, 2007.

[56] K. Perumalla. Scaling time warp-based discrete event execution to 104 processors on a blue gene supercomputer. In *Proc. of the ACM Conference on Computing Frontiers (CF)*, 2007.

[57] P. Peschlow, T. Honecker, and P. Martini. A flexible dynamic partitioning algorithm for optimistic distributed simulation. In *Proc. of 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2007.

[58] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, N. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the cell broadband engine. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[59] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proc. of ACM/IEEE Confernece on Supercomputing*, page 55. ACM, 2003.

[60] M. Quinn. *Parallel Computing: Theory and Practice*. McGraw Hill, 1994.

[61] U. K. V. Rajasekaran, M. Chetlur, G. D. Sharma, R. Radhakrishnan, and P. A. Wilsey. Addressing communication latency issues on clusters for fine grained asynchronous applications — a case study. In *International Workshop on Personal Computer Based Network of Workstations*, PC-NOW'99, April 1999.

[62] P. L. Reiher, F. Wieland, and D. R. Jefferson. Limitation of optimism in the Time Warp operating system. In *Winter Simulation Conference*, pages 765–770. Society for Computer Simulation, December 1989.

[63] P. Reynolds and P. Dickens. SPECTRUM: A parallel simulation testbed. In *The Hypercube Conference*, 1989.

[64] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, April 1997.

[65] V. Sachdev, M. Hybinette, and E. Kraemer. Controlling over-optimism in time-warp via cpubased flow control. In *Proceedings of the 2004 Winter Simulation Conference*. IEEE, 2004.

[66] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proc. of the SIGPLAN Symposium on Compiler construction*, 1986.

[67] G. D. Sharma, N. B. Abu-Ghazaleh, U. V. Rajasekaran, and P. A. Wilsey. Optimizing message delivery in asynchronous distributed applications. In *11th International Conference on Parallel and Distributed Computing Systems*, September 1998.

[68] K. H. Shum. Replicating parallel simulation on heterogeneous clusters. *Journal of Systems Architecture*, 44:273–292, 1998.

[69] S. C. Tay, Y. M. Teo, and S. T. Kong. Speculative parallel simulation with an adaptive throttle scheme. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 116–123. IEEE, 1997.

[70] Sunil Thulasidasan, Shiva P. Kasiviswanathan, Stephan Eidenbenz, and Phillip Romero. Explicit spatial scattering for load balancing in conservatively synchronized parallel discrete event simulations. In *Proc. of 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2010.

[71] Tilera TILE64 processor, 2008. Documentation from Tilera Website http://www. tilera.com.

[72] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and finegrained parallel applications. In *Proc. of ACM/IEEE Confernece on Supercomputing*, pages 303–312. ACM, 2005.

[73] R. Vitali, A. Pellegrini, and F. Quaglia. Assessing load-sharing within optimistic simulation platforms. In *Proceedings of the 2012 Winter Simulation Conference*. IEEE, 2012.

[74] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 221–220. IEEE, 2012.

[75] WarpIV Technologies (J. Steinman et. al.). The warpiv parallel simulation kernel version 1.5.2, 2008. Software available from http://www.warpiv.com/.

[76] L. Wilson and D. Nicol. Experiments in automated load balancing. In *Proc. of the tenth workshop on Parallel and distributed simulation (PADS)*, pages 4–11, 1996.

[77] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of ASPLOS*, pages 129–142. ACM, 2010.

APPENDIX A: Publications

1. Jingjing Wang, Nael Abu-Ghazaleh, Dmitry Ponomarev, "Interference-Resiliend PDES on Multi-core Systems: Towards Proportional Slowdown", *to appear in the ACM SIGSIM-PADS Conference, Montreal, Canada, May 2013.*

2. Jingjing Wang, Ketan Bahulkar, Dmitry Ponomarev, Nael Abu-Ghazaleh "Can PDES Scale in Environments with Heterogeneous Delays?", *to appear in the ACM SIGSIM-PADS Conference, Montreal, Canada, May 2013.*

3. Deepak Jagtap, Nael Abu-Ghazaleh, Dmitry Ponomarev, "Optimization of Parallel Discrete Event Simulator for Multicore Systems", *International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

4. Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, Nael Abu-Ghazaleh, "Characterizing and Understanding PDES Behavior on Tilera Architecture", *26th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Zhangjiajie, China, July, 2012.

5. Ketan Bahulkar, Jingjing Wang, Nael Abu-Ghazaleh, Dmitry Ponomarev, "Partitioning on Dynamic Behavior for Parallel Discrete Event Simulation", *26th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Zhangjiajie, China, July, 2012.

6. Jingjing Wang, Dmitry Ponomarev, Nael Abu-Ghazaleh, "Performance Analysis of a Multithreaded PDES Simulator on Multicore Clusters", *26th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Zhangjiajie, China, July, 2012.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ADM	adaptive dynamic mapping
API	application program interface
APTE	average processing time per event
CDN	Coherence Dynamic Network
CMs	cluster of multicores
CPU	central processing unit
DES	discrete event simulation
DIMM	dual in-line memory module
DM	dynamic mapping
DMA	direct memory access
EPC	event processing computational granularity
FM	fixed mapping
GB	gigabyte
GVT	global virtual time
I/O	input/output
IDN	I/O Dynamic Network
ISA	instruction set architecture
KB	kilobyte
LADM	locality-aware dynamic mapping
LIFO	last-in, first-out
LOP	latency-oblivious partitioning
LP	logical process
LSP	latency-sensitive partitioning
LVT	local virtual time
MAC	media access control
MCIP	message consolidation and infrequent polling
MDN	Memory Dynamic Network
MPI	message passing interface
NUMA	non-uniform memory access
OS	operating system
PCI-e	peripheral component interconnect express
PCS	personal communication services
PDES	parallel discrete event simulation
PE	processing element
ROSS	Rensselaer's Optimistic Simulation Syst
ROSS-CMT	CM based ROSS
ROSS-MPI	MPI based ROSS
ROSS-MT	multithreaded ROSS

SMT	simultaneous multi-threaded
STN	Static Network
TDN	Tile Dynamic Network
TMC	Tilera Multicore Components
UDN	User Dynamic Network
VLIW	very long instruction word
VLSI	very-large-scale integration